

FDPS仕様書

谷川衝, 岩澤全規, 細野七月, 似鳥啓吾, 村主崇行, 行方大輔, 野村昴太郎, and
牧野淳一郎

理化学研究所 計算科学研究センター 粒子系シミュレータ研究チーム

目次

1	この文書の概要	18
2	FDPS 概要	19
2.1	開発目的	19
2.2	基本的な考えかた	19
2.2.1	大規模並列粒子シミュレーションの手順	19
2.2.2	ユーザーとFDPSの役割分担	20
2.2.3	ユーザーのやること	20
2.2.4	補足	21
2.3	コードの動作	21
3	ファイル構成	23
3.1	概要	23
3.2	ドキュメント	23
3.3	ソースファイル	23
3.3.1	拡張機能	23
3.3.1.1	Particle Mesh	23
3.3.1.2	Particle Mesh Multipole Method	23
3.3.1.3	x86 版 Phantom-GRAPe	24
3.3.1.3.1	低精度 N 体シミュレーション用	24
3.3.1.3.2	低精度カットオフ付き相互作用計算用	24
3.3.1.3.3	高精度 N 体シミュレーション用	24
3.4	テストコード	24
3.5	サンプルコード	24
3.5.1	重力 N 体シミュレーション	25
3.5.2	SPH シミュレーション	25

4	コンパイル時のマクロによる選択	26
4.1	概要	26
4.2	座標系	26
4.2.1	概要	26
4.2.2	直角座標系 3 次元	26
4.2.3	直角座標系 2 次元	26
4.3	並列処理	26
4.3.1	概要	26
4.3.2	OpenMP の使用	26
4.3.3	MPI の使用	26
4.4	データ型の精度	27
4.4.1	概要	27
4.4.2	既存の SuperParticleJ クラスと Moment クラスの精度	27
4.5	ツリーの最大深さ (レベル) の変更	27
4.6	粒子のソートの方法の変更	27
5	名前空間	28
5.1	概要	28
5.2	ParticleSimulator	28
5.2.1	ParticleMesh	28
5.2.2	ParticleMeshMultipoleMethod	28
6	データ型	30
6.1	概要	30
6.2	整数型	30
6.2.1	概要	30
6.2.2	PS::S32	30
6.2.3	PS::S64	30
6.2.4	PS::U32	30
6.2.5	PS::U64	31
6.2.6	PS::Count_t	31
6.3	実数型	31
6.3.1	概要	31
6.3.2	PS::F32	31
6.3.3	PS::F64	31
6.4	ベクトル型	32
6.4.1	概要	32
6.4.2	PS::Vector2	32
6.4.2.1	コンストラクタ	33
6.4.2.2	コピーコンストラクタ	34
6.4.2.3	メンバ変数	34

6.4.2.4	代入演算子	34
6.4.2.5	[] 演算子	35
6.4.2.6	加減算	36
6.4.2.7	ベクトルスカラ積	37
6.4.2.8	内積、外積	38
6.4.2.9	Vector2<U> への型変換	38
6.4.3	PS::Vector3	39
6.4.3.1	コンストラクタ	40
6.4.3.2	コピーコンストラクタ	41
6.4.3.3	メンバ変数	41
6.4.3.4	代入演算子	41
6.4.3.5	[] 演算子	42
6.4.3.6	加減算	43
6.4.3.7	ベクトルスカラ積	44
6.4.3.8	内積、外積	45
6.4.3.9	Vector3<U> への型変換	45
6.4.4	ベクトル型のラッパー	46
6.5	オルソトープ型	46
6.5.1	概要	46
6.5.2	PS::Orthotope2	47
6.5.2.1	メンバ変数	49
6.5.2.2	コンストラクタ	49
6.5.2.3	コピーコンストラクタ	50
6.5.2.4	初期化	50
6.5.2.5	結合操作	51
6.5.3	PS::Orthotope3	51
6.5.3.1	メンバ変数	54
6.5.3.2	コンストラクタ	54
6.5.3.3	コピーコンストラクタ	55
6.5.3.4	初期化	55
6.5.3.5	結合操作	56
6.5.4	オルソトープ型のラッパー	56
6.6	対称行列型	57
6.6.1	概要	57
6.6.2	PS::MatrixSym2	57
6.6.2.1	コンストラクタ	58
6.6.2.2	コピーコンストラクタ	59
6.6.2.3	代入演算子	60
6.6.2.4	加減算	60
6.6.2.5	トレースの計算	61

6.6.2.6	MatrixSym2<U> への型変換	62
6.6.3	PS::MatrixSym3	62
6.6.3.1	コンストラクタ	63
6.6.3.2	コピーコンストラクタ	64
6.6.3.3	代入演算子	65
6.6.3.4	加減算	65
6.6.3.5	トレースの計算	66
6.6.3.6	MatrixSym3<U> への型変換	67
6.6.4	対称行列型のラッパー	67
6.7	PS::SEARCH_MODE 型	68
6.7.1	概要	68
6.7.2	PS::SEARCH_MODE_LONG	68
6.7.3	PS::SEARCH_MODE_LONG_CUTOFF	68
6.7.4	PS::SEARCH_MODE_GATHER	68
6.7.5	PS::SEARCH_MODE_SCATTER	69
6.7.6	PS::SEARCH_MODE_SYMMETRY	69
6.7.7	PS::SEARCH_MODE_LONG_SCATTER	69
6.7.8	PS::SEARCH_MODE_LONG_SYMMETRY	69
6.7.9	PS::SEARCH_MODE_LONG_CUTOFF_SCATTER	69
6.7.10	PS::SEARCH_MODE_LONG_PARTICLE_MESH_MULTIPOLE	69
6.8	列挙型	69
6.8.1	概要	69
6.8.2	PS::BOUNDARY_CONDITION 型	70
6.8.2.1	概要	70
6.8.2.2	PS::BOUNDARY_CONDITION_OPEN	70
6.8.2.3	PS::BOUNDARY_CONDITION_PERIODIC_X	70
6.8.2.4	PS::BOUNDARY_CONDITION_PERIODIC_Y	70
6.8.2.5	PS::BOUNDARY_CONDITION_PERIODIC_Z	70
6.8.2.6	PS::BOUNDARY_CONDITION_PERIODIC_XY	71
6.8.2.7	PS::BOUNDARY_CONDITION_PERIODIC_XZ	71
6.8.2.8	PS::BOUNDARY_CONDITION_PERIODIC_YZ	71
6.8.2.9	PS::BOUNDARY_CONDITION_PERIODIC_XYZ	71
6.8.2.10	PS::BOUNDARY_CONDITION_SHEARING_BOX	71
6.8.2.11	PS::BOUNDARY_CONDITION_USER_DEFINED	71
6.8.3	PS::INTERACTION_LIST_MODE 型	71
6.8.3.1	概要	71
6.8.3.2	PS::MAKE_LIST	72
6.8.3.3	PS::MAKE_LIST_FOR_REUSE	72
6.8.3.4	PS::REUSE_INTERACTION_LIST	72
6.8.4	PS::EXCHANGE_LET_MODE 型	72

6.8.4.1	概要	72
6.8.4.2	PS::EXCHANGE_LET_A2A	72
6.8.4.3	PS::EXCHANGE_LET_P2P_EXACT	73
6.8.4.4	EXCHANGE_LET_P2P_FAST	73
6.8.5	PS::CALC_DISTANCE_TYPE 型	73
6.8.5.1	概要	73
6.8.5.2	CALC_DISTANCE_TYPE_NORMAL	73
6.8.5.3	CALC_DISTANCE_TYPE_NEAREST_X	73
6.8.5.4	CALC_DISTANCE_TYPE_NEAREST_Y	74
6.8.5.5	CALC_DISTANCE_TYPE_NEAREST_Z	74
6.8.5.6	CALC_DISTANCE_TYPE_NEAREST_XY	74
6.8.5.7	CALC_DISTANCE_TYPE_NEAREST_XZ	74
6.8.5.8	CALC_DISTANCE_TYPE_NEAREST_YZ	74
6.8.5.9	CALC_DISTANCE_TYPE_NEAREST_XYZ	74
6.9	PS::TimeProfile	74
6.9.1	概要	74
6.9.1.1	メンバ変数	76
6.9.1.2	加算	77
6.9.1.3	除算	77
6.9.1.4	乗算	78
6.9.1.5	縮約	78
6.9.1.6	初期化	78
6.9.1.7	出力	79
7	ユーザー定義クラス・ユーザー定義関数オブジェクト	80
7.1	概要	80
7.2	FullParticle クラス	80
7.2.1	概要	80
7.2.2	前提	81
7.2.3	必要なメンバ関数	81
7.2.3.1	概要	81
7.2.3.2	FP::getPos	81
7.2.3.3	FP::copyFromForce	82
7.2.4	場合によっては必要なメンバ関数	82
7.2.4.1	概要	82
7.2.4.2	相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合	82
7.2.4.2.1	FP::getRSearch	82
7.2.4.3	粒子群クラスのファイル入出力 API を用いる場合	83
7.2.4.3.1	FP::readAscii	83
7.2.4.3.2	FP::writeAscii	84

7.2.4.3.3	FP::readBinary	84
7.2.4.3.4	FP::writeBinary	85
7.2.4.4	ParticleSystem::adjustPositionIntoRootDomain を用いる場合	85
7.2.4.4.1	FP::setPos	85
7.2.4.5	Particle Mesh クラスを用いる場合	85
7.2.4.5.1	FP::getChargeParticleMesh	86
7.2.4.5.2	FP::copyFromForceParticleMesh	86
7.2.4.6	粒子交換時に粒子データをシリアルライズして送る場合	86
7.2.4.6.1	FP::pack	87
7.2.4.6.2	FP::unPack	87
7.3	EssentialParticleI クラス	88
7.3.1	概要	88
7.3.2	前提	88
7.3.3	必要なメンバ関数	88
7.3.3.1	概要	88
7.3.3.2	EPI::getPos	88
7.3.3.3	EPI::copyFromFP	89
7.3.4	場合によっては必要なメンバ関数	89
7.3.4.1	概要	89
7.3.4.2	相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合	90
7.3.4.2.1	EPI::getRSearch	90
7.4	EssentialParticleJ クラス	90
7.4.1	概要	90
7.4.2	前提	90
7.4.3	必要なメンバ関数	91
7.4.3.1	概要	91
7.4.3.2	EPJ::getPos	91
7.4.3.3	EPJ::copyFromFP	91
7.4.4	場合によっては必要なメンバ関数	92
7.4.4.1	概要	92
7.4.4.2	相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合	92
7.4.4.2.1	EPJ::getRSearch	92
7.4.4.3	BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を用いる場合	93
7.4.4.3.1	EPJ::setPos	93
7.4.4.4	粒子の id 番号から対応する EPJ を取得したい場合	93
7.4.4.4.1	EPJ::getId	93

7.4.4.5	LET 交換時に粒子データをシリアルライズして送る場合	94
7.4.4.5.1	EPJ::pack	94
7.4.4.5.2	EPJ::unPack	94
7.5	Moment クラス	95
7.5.1	概要	95
7.5.2	既存のクラス	95
7.5.2.1	概要	95
7.5.2.2	PS::SEARCH_MODE_LONG	95
7.5.2.2.1	PS::MomentMonopole	95
7.5.2.2.2	PS::MomentQuadrupole	96
7.5.2.2.3	PS::MomentMonopoleGeometricCenter	97
7.5.2.2.4	PS::MomentDipoleGeometricCenter	97
7.5.2.2.5	PS::MomentQuadrupoleGeometricCenter	98
7.5.2.3	PS::SEARCH_MODE_LONG_SCATTER	99
7.5.2.3.1	PS::MomentMonopoleScatter	99
7.5.2.3.2	PS::MomentQuadrupoleScatter	100
7.5.2.4	PS::SEARCH_MODE_LONG_SYMMETRY	100
7.5.2.4.1	PS::MomentMonopoleSymmetry	100
7.5.2.4.2	PS::MomentQuadrupoleSymmetry	101
7.5.2.5	PS::SEARCH_MODE_LONG_CUTOFF	102
7.5.2.5.1	PS::MomentMonopoleCutoff	102
7.5.3	必要なメンバ関数	103
7.5.3.1	概要	103
7.5.3.2	コンストラクタ	103
7.5.3.3	Mom::init	104
7.5.3.4	Mom::getPos	104
7.5.3.5	Mom::getCharge	105
7.5.3.6	Mom::getVertexIn	105
7.5.3.7	Mom::accumulateAtLeaf	105
7.5.3.8	Mom::accumulate	106
7.5.3.9	Mom::set	106
7.5.3.10	Mom::accumulateAtLeaf2	107
7.5.3.11	Mom::accumulate2	107
7.5.4	場合によっては必要なメンバ関数	108
7.5.4.1	概要	108
7.5.4.2	相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG_CUTOFF、 PS::SEARCH_MODE_LONG_SCATTER、 PS::SEARCH_MODE_LONG_SYMMETRY のいずれかを用いる場合	108

7.5.4.3	Mom::getVertexIn	108
7.5.4.4	Mom::getVertexOut	109
7.6	SuperParticleJ クラス	110
7.6.1	概要	110
7.6.2	既存のクラス	111
7.6.2.1	PS::SEARCH_MODE_LONG	111
7.6.2.1.1	PS::SPJMonopole	111
7.6.2.1.2	PS::SPJQuadrupole	111
7.6.2.1.3	PS::SPJMonopoleGeometricCenter	112
7.6.2.1.4	PS::SPJDipoleGeometricCenter	113
7.6.2.1.5	PS::SPJQuadrupoleGeometricCenter	113
7.6.2.2	PS::SEARCH_MODE_LONG_SCATTER	114
7.6.2.2.1	PS::SPJMonopoleScatter	114
7.6.2.2.2	PS::SPJQuadrupoleScatter	114
7.6.2.3	PS::SEARCH_MODE_LONG_SYMMETRY	115
7.6.2.3.1	PS::SPJMonopoleSymmetry	115
7.6.2.3.2	PS::SPJQuadrupoleSymmetry	116
7.6.2.4	PS::SEARCH_MODE_LONG_CUTOFF	116
7.6.2.4.1	PS::SPJMonopoleCutoff	116
7.6.3	必要なメンバ関数	117
7.6.3.1	概要	117
7.6.3.2	SPJ::getPos	117
7.6.3.3	SPJ::setPos	117
7.6.3.4	SPJ::copyFromMoment	118
7.6.3.5	SPJ::convertToMoment	118
7.6.3.6	SPJ::clear	119
7.6.4	場合によっては必要なメンバ関数	119
7.6.4.1	概要	119
7.6.4.2	LET 交換時に粒子データをシリアルライズして送る場合	119
7.6.4.2.1	SPJ::pack	120
7.6.4.2.2	SPJ::unPack	120
7.7	Force クラス	121
7.7.1	概要	121
7.7.2	前提	121
7.7.3	必要なメンバ関数	121
7.7.3.1	Result::clear	121
7.8	ヘッダクラス	121
7.8.1	概要	121
7.8.2	前提	122
7.8.3	場合によっては必要なメンバ関数	122

7.8.3.1	Hdr::readAscii	122
7.8.3.2	Hdr::writeAscii	122
7.8.3.3	Hdr::readBinary	123
7.8.3.4	Hdr::writeBinary	123
7.9	関数オブジェクト calcForceEpEp	123
7.9.1	概要	123
7.9.2	前提	124
7.9.3	gravityEpEp::operator ()	124
7.10	関数オブジェクト calcForceSpEp	124
7.10.1	概要	124
7.10.2	前提	125
7.10.3	gravitySpEp::operator ()	125
7.11	関数オブジェクト calcForceDispatch	126
7.11.1	概要	126
7.11.2	短距離力の場合	126
7.11.3	長距離力の場合	127
7.12	関数オブジェクト calcForceRetrieve	128
7.12.1	概要	128
8	プログラムの開始と終了	129
8.1	概要	129
8.2	API	129
8.2.1	PS::Initialize	129
8.2.2	PS::Finalize	129
8.2.3	PS::Abort	130
8.2.4	PS::DisplayInfo	130
9	モジュール	131
9.1	標準機能	131
9.1.1	概要	131
9.1.2	領域クラス	131
9.1.2.1	オブジェクトの生成	131
9.1.2.2	API	131
9.1.2.2.1	初期設定	131
9.1.2.2.1.1	コンストラクタ	132
9.1.2.2.1.2	PS::DomainInfo::initialize	133
9.1.2.2.1.3	PS::DomainInfo::setCommInfo	133
9.1.2.2.1.4	PS::DomainInfo::setNumberOfDomainMultiDimension	134
9.1.2.2.1.5	PS::DomainInfo::setDomain	134
9.1.2.2.1.6	PS::DomainInfo::setBoundaryCondition	135
9.1.2.2.1.7	PS::DomainInfo::getBoundaryCondition	135

9.1.2.2.1.8	PS::DomainInfo::setPosRootDomain	135
9.1.2.2.1.9	PS::DomainInfo::setPosRootDomainX	136
9.1.2.2.1.10	PS::DomainInfo::setPosRootDomainY	137
9.1.2.2.1.11	PS::DomainInfo::setPosRootDomainZ	137
9.1.2.2.2	領域分割	137
9.1.2.2.2.1	PS::DomainInfo::collectSampleParticle	138
9.1.2.2.2.2	PS::DomainInfo::decomposeDomain	139
9.1.2.2.2.3	PS::DomainInfo::decomposeDomainAll	140
9.1.2.2.3	時間計測	141
9.1.2.2.3.1	PS::DomainInfo::getTimeProfile	141
9.1.2.2.3.2	PS::DomainInfo::clearTimeProfile	141
9.1.2.2.4	情報取得	142
9.1.2.2.4.1	PS::DomainInfo::getUsedMemorySize	142
9.1.3	粒子群クラス	142
9.1.3.1	オブジェクトの生成	143
9.1.3.2	API	143
9.1.3.2.1	初期設定	143
9.1.3.2.1.1	コンストラクタ	144
9.1.3.2.1.2	PS::ParticleSystem::initialize	144
9.1.3.2.1.3	PS::ParticleSystem::setCommInfo	144
9.1.3.2.1.4	PS::ParticleSystem:: setAverateTargetNumberOfSampleParticlePerProcess	145
9.1.3.2.2	情報取得	145
9.1.3.2.2.1	PS::ParticleSystem::operator []	146
9.1.3.2.2.2	PS::ParticleSystem::getNumberOfParticleLocal	146
9.1.3.2.2.3	PS::ParticleSystem::getNumberOfParticleGlobal	147
9.1.3.2.2.4	PS::DomainInfo::getUsedMemorySize	147
9.1.3.2.3	ファイル入出力	147
9.1.3.2.3.1	PS::ParticleSystem::readParticleAscii	148
9.1.3.2.3.2	PS::ParticleSystem::readParticleBinary	150
9.1.3.2.3.3	PS::ParticleSystem::writeParticleAscii	151
9.1.3.2.3.4	PS::ParticleSystem::writeParticleBinary	153
9.1.3.2.4	粒子交換	155
9.1.3.2.4.1	PS::ParticleSystem::exchangeParticle	156
9.1.3.2.5	粒子の追加、削除	156
9.1.3.2.5.1	PS::ParticleSystem::addOneParticle()	156
9.1.3.2.5.2	PS::ParticleSystem::removeParticle()	157
9.1.3.2.6	時間計測	157
9.1.3.2.6.1	PS::ParticleSystem::getTimeProfile	157
9.1.3.2.6.2	PS::ParticleSystem::clearTimeProfile	158

9.1.3.2.7	その他	158
9.1.3.2.7.1	PS::ParticleSystem::adjustPositionIntoRootDomain	159
9.1.3.2.7.2	PS::ParticleSystem::setNumberOfParticleLocal	159
9.1.3.2.7.3	PS::ParticleSystem::sortParticle	160
9.1.4	相互作用ツリークラス	160
9.1.4.1	オブジェクトの生成	160
9.1.4.1.1	PS::SEARCH_MODE_LONG	161
9.1.4.1.2	PS::SEARCH_MODE_LONG_SCATTER	162
9.1.4.1.3	PS::SEARCH_MODE_LONG_SYMMETRY	163
9.1.4.1.4	PS::SEARCH_MODE_LONG_CUTOFF	163
9.1.4.1.5	PS::SEARCH_MODE_GATHER	164
9.1.4.1.6	PS::SEARCH_MODE_SCATTER	164
9.1.4.1.7	PS::SEARCH_MODE_SYMMETRY	164
9.1.4.2	API	165
9.1.4.2.1	初期設定	166
9.1.4.2.1.1	コンストラクタ	166
9.1.4.2.1.2	PS::TreeForForce::initialize	167
9.1.4.2.1.3	PS::TreeForForce::setCommInfo	167
9.1.4.2.1.4	PS::TreeForForce::setExchangeLETMode	168
9.1.4.2.2	低レベル関数	168
9.1.4.2.2.1	PS::TreeForForce::calcForce	169
9.1.4.2.2.2	PS::TreeForForce::getForce	170
9.1.4.2.3	高レベル関数	170
9.1.4.2.3.1	PS::TreeForForce::calcForceAllAndWriteBack	172
9.1.4.2.3.2	PS::TreeForForce::calcForceAll	174
9.1.4.2.3.3	PS::TreeForForce::calcForceAllAndWriteBackMultiWalk	175
9.1.4.2.3.4	PS::TreeForForce::calcForceAllMultiWalk	177
9.1.4.2.3.5	PS::TreeForForce::calcForceAllAndWriteBackMultiWalkIndex	177
9.1.4.2.3.6	PS::TreeForForce::calcForceAllMultiWalkIndex	180
9.1.4.2.3.7	PS::TreeForForce::calcForceAndWriteBack	180
9.1.4.2.4	ネイバーリスト	181
9.1.4.2.4.1	getNeighborListOneParticle	182
9.1.4.2.5	時間計測	183
9.1.4.2.5.1	PS::TreeForForce::getTimeProfile	183
9.1.4.2.5.2	PS::TreeForForce::clearTimeProfile	184
9.1.4.2.6	情報取得	184
9.1.4.2.6.1	PS::TreeForForce::getNumberOfInteractionEPEPLocal185	
9.1.4.2.6.2	PS::TreeForForce::getNumberOfInteractionEPEPGlobal185	
9.1.4.2.6.3	PS::TreeForForce::getNumberOfInteractionEPSPLocal186	
9.1.4.2.6.4	PS::TreeForForce::getNumberOfInteractionEPSPGlobal186	

9.1.4.2.6.5	PS::TreeForForce::clearNumberOfInteraction	187
9.1.4.2.6.6	PS::TreeForForce::getNumberOfWalkLocal	187
9.1.4.2.6.7	PS::TreeForForce::getNumberOfWalkGlobal	187
9.1.4.2.6.8	PS::TreeForForce::getUsedMemorySize	188
9.1.4.2.7	粒子 id から EPJ を取得する	188
9.1.4.2.7.1	getEpjFromId	188
9.1.4.2.8	Particle Mesh Mulipole Method の API	189
9.1.4.2.8.1	setParamsPMMM	189
9.1.5	通信用データクラス	189
9.1.5.1	オブジェクトの生成	189
9.1.5.2	API	190
9.1.5.2.1	初期設定	192
9.1.5.2.1.1	コンストラクタ	192
9.1.5.2.2	PS::CommInfo::setCommunicator	192
9.1.5.2.2.1	PS::CommInfo::create	193
9.1.5.2.2.2	PS::CommInfo::split	193
9.1.5.2.3	情報取得	193
9.1.5.2.3.1	PS::CommInfo::getRank	194
9.1.5.2.3.2	PS::CommInfo::getNumberOfProc	194
9.1.5.2.3.3	PS::CommInfo::getRankMultiDim	194
9.1.5.2.3.4	PS::CommInfo::getNumberOfProcMultiDim	194
9.1.5.2.3.5	PS::CommInfo::synchronizeConditionalBranchAND	195
9.1.5.2.3.6	PS::CommInfo::synchronizeConditionalBranchOR	195
9.1.5.2.3.7	PS::CommInfo::getMinValue	195
9.1.5.2.3.8	PS::CommInfo::getMaxValue	196
9.1.5.2.3.9	PS::CommInfo::getSum	197
9.1.5.2.3.10	PS::CommInfo::broadcast	197
9.1.5.2.3.11	PS::Comm::getRank	198
9.1.5.2.3.12	PS::Comm::getNumberOfProc	198
9.1.5.2.3.13	PS::Comm::getRankMultiDim	198
9.1.5.2.3.14	PS::Comm::getNumberOfProcMultiDim	199
9.1.5.2.3.15	PS::Comm::synchronizeConditionalBranchAND	199
9.1.5.2.3.16	PS::Comm::synchronizeConditionalBranchOR	199
9.1.5.2.3.17	PS::Comm::getMinValue	200
9.1.5.2.3.18	PS::Comm::getMaxValue	200
9.1.5.2.3.19	PS::Comm::getSum	201
9.1.5.2.3.20	PS::Comm::broadcast	201
9.1.6	その他関数	202
9.1.6.1	時間計測	202
9.1.6.1.1	PS::GetWtime	202

9.2	拡張機能	202
9.2.1	概要	202
9.2.2	Particle Mesh クラス	202
9.2.2.1	オブジェクトの生成	203
9.2.2.2	API	203
9.2.2.2.1	初期設定	203
9.2.2.2.1.1	コンストラクタ	204
9.2.2.2.2	低レベル API	204
9.2.2.2.2.1	PS::PM::ParticleMesh::setDomainInfoParticleMesh	205
9.2.2.2.2.2	PS::PM::ParticleMesh::setParticleParticleMesh	205
9.2.2.2.2.3	PS::PM::ParticleMesh::calcMeshForceOnly	205
9.2.2.2.2.4	PS::PM::ParticleMesh::getForce	206
9.2.2.2.2.5	PS::PM::ParticleMesh::getPotential	206
9.2.2.2.3	高レベル API	207
9.2.2.2.3.1	PS::PM::ParticleMesh::calcForceAllAndWriteBack	207
9.2.2.3	使用済マクロ	208
9.2.2.4	Particle Mesh クラスの使いかた	209
9.2.2.4.1	Particle Mesh クラスのコンパイル	210
9.2.2.4.2	FDPS コードを記述	210
9.2.2.4.3	FDPS コードのコンパイル	210
9.2.2.4.4	注意事項	210
9.2.3	x86 版 phantom-GRAPE	211
9.2.4	Particle Mesh Multipole Moment クラス	211
9.2.4.1	オブジェクトの生成	211
9.2.4.2	API	212
9.2.4.2.1	初期設定	212
9.2.4.2.1.1	コンストラクタ	212
9.2.4.2.1.2	PS::PMM::ParticleMeshMultipole::initialize	212
9.2.4.2.1.3	PS::PMM::ParticleMeshMultipole::calcForceAllAndWriteBack	213
9.2.4.3	ParticleMeshMultipole クラスの使いかた	213
10	エラー検出	215
10.1	概要	215
10.2	コンパイル時のエラー	215
10.3	実行時のエラー	215
10.3.1	PS_ERROR: can not open input file	215
10.3.2	PS_ERROR: can not open output file	216
10.3.3	PS_ERROR: Do not initialize the tree twice	216
10.3.4	PS_ERROR: The opening criterion of the tree must be ≥ 0.0	216
10.3.5	PS_ERROR: The limit number of the particles in the leaf cell must be > 0	216

10.3.6	PS_ERROR: The limit number of particles in ip groups msut be >= that in leaf cells	217
10.3.7	PS_ERROR: The number of particles of this process is beyond the FDPS limit number	217
10.3.8	PS_ERROR: The forces w/o cutoff can be evaluated only under the open boundary condition	217
10.3.9	PS_ERROR: A particle is out of root domain	218
10.3.10	PS_ERROR: The smoothing factor of an exponential moving average is must between 0 and 1.	218
10.3.11	PS_ERROR: The coodinate of the root domain is inconsistent.	218
10.3.12	PS_ERROR: Vector invalid accesse	218
11	よく知られているバグ	219
12	限界	220
13	ユーザーサポート	221
13.1	コンパイルできない場合	221
13.2	コードがうまく動かない場合	221
13.3	その他	221
14	ライセンス	222
15	変更履歴	223
A	ユーザー定義クラスの実装例	228
A.1	FullParticle クラス	228
A.1.1	概要	228
A.1.2	前提	228
A.1.3	必要なメンバ関数	228
A.1.3.1	概要	228
A.1.3.2	FP::getPos	228
A.1.3.3	FP::copyFromForce	229
A.1.4	場合によっては必要なメンバ関数	230
A.1.4.1	概要	230
A.1.4.2	相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合	230
A.1.4.2.1	FP::getRSearch	230
A.1.4.3	粒子群クラスのファイル入出力 API を用いる場合	231
A.1.4.3.1	FP::readAscii	231
A.1.4.3.2	FP::writeAscii	232
A.1.4.3.3	FP::readBinary	233

A.1.4.3.4	FP::writeBinary	234
A.1.4.4	ParticleSystem::adjustPositionIntoRootDomain を用いる場合	235
A.1.4.4.1	FP::setPos	235
A.1.4.5	Particle Mesh クラスを用いる場合	235
A.1.4.5.1	FP::getChargeParticleMesh	236
A.1.4.5.2	FP::copyFromForceParticleMesh	236
A.2	EssentialParticleI クラス	237
A.2.1	概要	237
A.2.2	前提	237
A.2.3	必要なメンバ関数	237
A.2.3.1	概要	237
A.2.3.2	EPI::getPos	238
A.2.3.3	EPI::copyFromFP	239
A.2.4	場合によっては必要なメンバ関数	240
A.2.4.1	概要	240
A.2.4.2	相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合	240
A.2.4.2.1	EPI::getRSearch	240
A.3	EssentialParticleJ クラス	241
A.3.1	概要	241
A.3.2	前提	241
A.3.3	必要なメンバ関数	241
A.3.3.1	概要	241
A.3.3.2	EPJ::getPos	241
A.3.3.3	EPJ::copyFromFP	242
A.3.4	場合によっては必要なメンバ関数	243
A.3.4.1	概要	243
A.3.4.2	相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合	243
A.3.4.2.1	EPJ::getRSearch	243
A.3.4.3	BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を用いる場合	244
A.3.4.3.1	EPJ::setPos	244
A.4	Moment クラス	245
A.4.1	概要	245
A.4.2	既存のクラス	245
A.4.2.1	概要	245
A.4.2.2	PS::SEARCH_MODE_LONG	245
A.4.2.2.1	PS::MomentMonopole	246

A.4.2.2.2	PS::MomentQuadrupole	246
A.4.2.2.3	PS::MomentMonopoleGeometricCenter	247
A.4.2.2.4	PS::MomentDipoleGeometricCenter	247
A.4.2.2.5	PS::MomentQuadrupoleGeometricCenter	248
A.4.2.3	PS::SEARCH_MODE_LONG_CUTOFF	249
A.4.2.3.1	PS::MomentMonopoleCutoff	249
A.4.3	必要なメンバ関数	250
A.4.3.1	概要	250
A.4.3.2	コンストラクタ	250
A.4.3.3	Mom::init	251
A.4.3.4	Mom::getPos	252
A.4.3.5	Mom::getCharge	253
A.4.3.6	Mom::accumulateAtLeaf	253
A.4.3.7	Mom::accumulate	254
A.4.3.8	Mom::set	255
A.4.3.9	Mom::accumulateAtLeaf2	256
A.4.3.10	Mom::accumulate2	257
A.5	SuperParticleJ クラス	258
A.5.1	概要	258
A.5.2	既存のクラス	258
A.5.2.1	PS::SEARCH_MODE_LONG	258
A.5.2.1.1	PS::SPJMonopole	258
A.5.2.1.2	PS::SPJQuadrupole	259
A.5.2.1.3	PS::SPJMonopoleGeometricCenter	259
A.5.2.1.4	PS::SPJDipoleGeometricCenter	260
A.5.2.1.5	PS::SPJQuadrupoleGeometricCenter	261
A.5.2.2	PS::SEARCH_MODE_LONG_CUTOFF	261
A.5.2.2.1	PS::SPJMonopoleCutoff	261
A.5.3	必要なメンバ関数	262
A.5.3.1	概要	262
A.5.3.2	SPJ::getPos	262
A.5.3.3	SPJ::setPos	263
A.5.3.4	SPJ::copyFromMoment	264
A.5.3.5	SPJ::convertToMoment	265
A.5.3.6	SPJ::clear	266
A.6	Force クラス	266
A.6.1	概要	266
A.6.2	前提	266
A.6.3	必要なメンバ関数	267
A.6.3.1	Result::clear	267

A.7	ヘッダクラス	267
A.7.1	概要	267
A.7.2	前提	268
A.7.3	場合によっては必要なメンバ関数	268
A.7.3.1	Hdr::readAscii	268
A.7.3.2	Hdr::writeAscii	269
A.7.3.3	Hdr::readBinary	269
A.7.3.4	Hdr::writeBinary	270
A.8	関数オブジェクト calcForceEpEp	271
A.8.1	概要	271
A.8.2	前提	271
A.8.3	gravityEpEp::operator ()	271
A.9	関数オブジェクト calcForceSpEp	273
A.9.1	概要	273
A.9.2	前提	273
A.9.3	gravitySpEp::operator ()	273
A.10	関数オブジェクト calcForceDispatch	275
A.10.1	概要	275
A.10.2	前提	275
A.10.3	例	275
A.11	関数オブジェクト calcForceRetrieve	280
A.11.1	概要	280
A.11.2	前提	281
B	Version7 系から Version8 系への大きな変更点	282

1 この文書の概要

この文書は大規模並列粒子シミュレーションの開発を支援する Framework for Developing Particle Simulator (FDPS) の仕様書である。この文書は理化学研究所計算科学研究機構粒子系シミュレータ研究チームの谷川衝、岩澤全規、細野七月、似鳥啓吾、村主崇行、牧野淳一郎によって記述された。

この文書は以下のような構成となっている。

節 2、3、4 には、FDPS を使ってプログラムを書く際に前提となる情報が記述されている。節 2 には、FDPS の概要として、FDPS の基本的な考えかたや動作が記述されている。節 3 には、FDPS のファイル構成が記述されている。節 4 には、FDPS の API を使用したコードをコンパイルする時にどのようなマクロを用いればよいかが記述されている。

節 5、6、7、8、9 には、FDPS を使ってプログラムを書く際に必要となる情報が提供されている。節 5 には、FDPS 内での名前空間の構造についてが記述されている。節 6 には、FDPS で独自に定義されているデータ型が記述されている。節 7 には、FDPS の API を使用する際にユーザーが定義する必要があるクラスや関数オブジェクトについて記述されている。節 8 には、FDPS を開始するときと終了するときと呼ぶ必要のある API について記述されている。節 9 には、FDPS にあるモジュールとその API について記述されている。

節 10、11、12、13 には、FDPS の API を使用したコードを記述したがコードが思ったように動作しない場合に有用な情報が記載されている。節 10 にはエラーメッセージについてが記述されている。節 11 には、よく知られているバグについて記述されている。節 12 には、FDPS の限界について記述されている。節 13 には、ユーザーサポートに関する情報が記述されている。

最後に節 14 には FDPS のライセンスに関する情報が、節 15 にはこの文書の変更履歴が記述されている。

2 FDPS 概要

この節ではFDPSの概要を記述する。FDPSの開発目的、FDPSの基本的な考えかた、FDPSを使用して作成したコードの動作について概説する。

2.1 開発目的

粒子シミュレーションは、重力 N 体シミュレーション、SPH シミュレーション、渦糸法、MPS 法、分子動力学シミュレーションなど理工学のような様々な分野で使用されている。より大きい空間スケール、より高い空間分解能 (または質量分解能)、より長い時間スケールの物理現象を追跡するために、高性能な粒子シミュレーションコードへの要請はますます強くなっている。

高性能な粒子シミュレーションコードを組むためには、シミュレーションコードの大規模並列化を避けることはできない。粒子シミュレーションコードの大規模並列化をする際には、ロードバランスのため動的領域分割、領域分割に合わせた粒子交換、ノード間通信の削減と最適化、キャッシュ利用効率の向上、SIMD ユニット利用効率の向上、アクセラレータへの対応など、数多くの困難な処理を行う必要がある。現在、研究グループは個別にこれらの処理へ対応している。

しかし、上記の処理は粒子シミュレーション共通のものである。FDPS の開発目的は、これらの処理を高速に行うライブラリを提供し、大規模並列化への対応に追われていた研究者の負担を軽くすることである。FDPS を使うことで、研究者がよりクリエイティブな仕事に専念できるようになれば、幸いである。

2.2 基本的な考えかた

ここではFDPSの基本的な考えかたについて記述する。

2.2.1 大規模並列粒子シミュレーションの手順

まずFDPSにおいて、大規模並列粒子シミュレーションがどのような手順で行われることを想定しているかを記述する。粒子シミュレーションは、以下のような微分方程式を時間発展させるものである。

$$\frac{d\mathbf{u}_i}{dt} = \sum_j f(\mathbf{u}_i, \mathbf{u}_j) + \sum_s g(\mathbf{u}_i, \mathbf{v}_s) \quad (1)$$

ここで \mathbf{u}_i は粒子 i の物理量ベクトルであり、この物理量には質量、位置、速度など粒子が持つあらゆる物理量が含まれる。関数 f は粒子 j から粒子 i への作用を規定する。以後、作用を受ける粒子を i 粒子、作用を与える粒子を j 粒子と呼ぶことにする。 \mathbf{v}_s は i 粒子から十分遠方にある粒子を1つの粒子としてまとめた粒子 (以後、この粒子を超粒子と呼ぶ) の物理量ベクトルである。関数 g は超粒子から i 粒子への作用を規定する。式 (1) の第2項は、重力

やクーロン力など無限遠まで到達する長距離力の場合はゼロではない。しかし流体の圧力のような短距離力はゼロである。

大規模並列化された粒子シミュレーションコードは以下の手順で式 (1) を時間発展させる。ここではデータの入出力や初期化は省略している。

1. 以下の 2 段階の手順でどのプロセスがどの粒子の式 (1) を時間発展させるか決める。
 - (a) プロセスの間でロードバランスを取れるように、シミュレーションで扱っている空間の領域を分割し、各プロセスの担当領域を決める (領域分割)。
 - (b) 各プロセスが、自分の担当する領域に存在する全粒子の物理量ベクトル u_i を持つように、他のプロセスと物理量ベクトル u_i を交換する (粒子交換)。
2. 各プロセスは、自分の担当する全粒子の式 (1) の右辺を計算するのに必要な j 粒子の物理量ベクトル u_j と超粒子の物理量ベクトル v_s を他のプロセスと通信することで集めて、 j 粒子のリストと超粒子のリスト (まとめて相互作用リストと呼ぶ) を作る (相互作用リストの作成)。
3. 各プロセスは自分の担当する全粒子に対して、式 (1) の右辺を計算し、 du_i/dt を求める (相互作用の計算)。
4. 各プロセスは、自分の担当する全粒子の物理量ベクトル u_i とその時間導関数 du_i/dt を使って、全粒子の時間積分を実行し、次の時刻の物理量ベクトル u_i を求める (時間積分)。
5. 手順 1 に戻る。

2.2.2 ユーザーと FDPS の役割分担

FDPS は、プロセス間の通信が発生する処理は FDPS が担当し、プロセス間の通信の発生しない処理はユーザーが担当するという役割分担を基本としている。従って、前節に挙げた、領域分割・粒子交換 (項目 1)・相互作用リストの作成 (項目 2) を FDPS が、相互作用の計算 (項目 3)・時間積分 (項目 4) をユーザーが担当することになる。ユーザーは FDPS の API を呼び出すだけで、大規模並列化に関わる煩雑な処理を避けつつ、高性能な任意の相互作用の粒子シミュレーションコードを手に入れることができる。

2.2.3 ユーザーのやること

ユーザーが FDPS を使って粒子シミュレーションコードを作成するときにやることは以下の項目である。

- 粒子の定義 (節 7)。粒子の持つ物理量 (式 (1) で言えば u_i) の指定。例えば質量、位置、速度、加速度、元素組成、粒子サイズ、など。

- 相互作用の定義 (節 7)。粒子間の相互作用 (式 (1) で言えば関数 f, g) を指定。例えば、重力、クーロン力、圧力、など。
- FDPS の API の呼出 (節 8, 9)

2.2.4 補足

式 (1) の右辺は 2 粒子間相互作用の重ね合わせである。従って、FDPS の API を呼ぶだけでは、3 つ以上の粒子の間の相互作用の計算を行うことはできない。しかし、FDPS はネイバーリストを返す API を用意している。ネイバーリストを用いれば、ユーザーはプロセス間の通信の処理をすることなく、このような相互作用の計算をできる。

節 2.2.1 で示した手順は、全粒子が同じ時間刻みを持っている。そのため、FDPS の API を呼び出すだけでは、独立時間刻みで時間積分を効率的に行うことができない。しかし、上と同じくネイバーリストを返す API があるため、Particle Particle Particle Tree 法を用いて独立時間刻みを実装することは可能であろう。

2.3 コードの動作

ここでは FDPS を使用して作成したコードの動作の概略を記述する。このコードには、4 つのモジュールがあることになる。3 つは FDPS のモジュールで、1 つはユーザー定義のモジュールである。まとめると以下ようになる。

- 領域クラス：全プロセスが担当する領域の情報と、領域分割を行う API を持つ
- 粒子群クラス：全粒子の情報と、プロセスの間での粒子交換を行う API を持つ
- 相互作用ツリークラス：粒子分布から作られたツリー構造と、相互作用リストを作成する API を持つ
- ユーザー定義クラス：ある 1 粒子を定義するクラス、粒子間の相互作用を定義する関数オブジェクトを持つ

これら 4 つのモジュールの間で情報がやり取りされる。これは図 1 で概観できる。図 1 に示された情報のやりとりは、節 2.2.1 に記述された手順 1 から 3 と、これらの手順以前に行われる手順 (手順 0 とする) に対応する。以下はこれらの手順の詳細な記述である。

0. ユーザー定義クラスのうち 1 粒子を定義するクラスが粒子群クラスへ、粒子間の相互作用を定義する関数オブジェクトが相互作用ツリークラスへ渡される。これはクラスの継承ではなく、粒子を定義するクラスは粒子群クラスのテンプレート引数として、粒子間の相互作用を定義する関数オブジェクトは相互作用ツリークラスの API の引数として渡される
1. 以下の 2 段階でロードバランスを取る

- (a) 領域クラスが持つ領域分割の API が呼ばれる。このとき粒子情報が粒子群クラスから領域クラスへ渡される (赤字と赤矢印)
 - (b) 粒子群クラスが持つ粒子交換の API が呼ばれる。このとき領域情報が領域クラスから粒子群クラスへ渡される (青字と青矢印)
2. 相互作用ツリークラスが持つ相互作用リストを作成する API が呼ばれる。このとき領域情報が領域クラスから相互作用ツリークラスへ、粒子情報が粒子群クラスから相互作用ツリークラスへ渡される (緑字と緑矢印)
3. 相互作用ツリークラスが持つ相互作用を定義した関数オブジェクトを呼び出す API が呼ばれる。相互作用計算が実行され、相互作用計算の結果が相互作用ツリークラスから粒子群クラスへ渡される (灰色の字と灰色矢印)

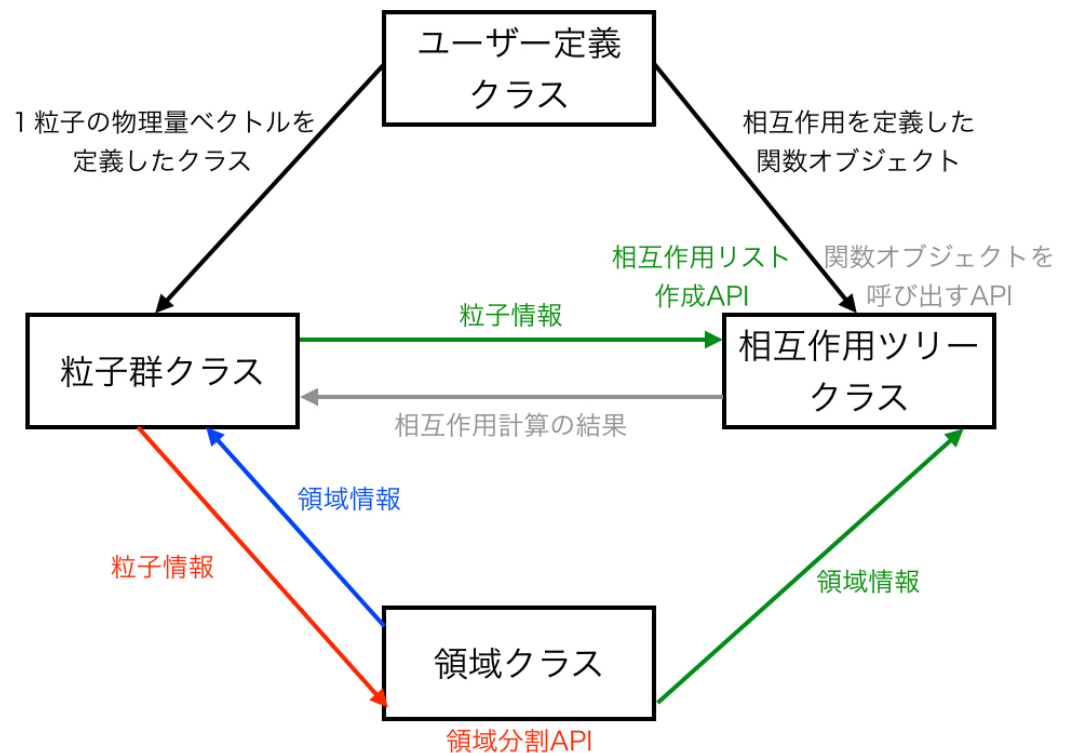


図 1: モジュールインターフェースと情報の流れの模式図。

3 ファイル構成

3.1 概要

ここではFDPSのファイル構成について記述する。ドキュメント、ソースファイル、テストコード、サンプルコードの順に記述する。

3.2 ドキュメント

ドキュメント関係のファイルはディレクトリ `doc` の下にある。チュートリアルが `doc_tutorial.pdf` であり、仕様書が `doc_specs.pdf` である。

3.3 ソースファイル

ソースファイルはディレクトリ `src` の下にある。標準機能関係のソースファイルは `src` の直下にある。ディレクトリ `src` の直下にあるヘッダファイル `particle_simulator.hpp` をソースファイルにインクルードすれば、FDPSの標準機能を使用できるようになる。

3.3.1 拡張機能

拡張機能関係のソースファイルはディレクトリ `src` の直下のディレクトリにそれぞれ入っている。拡張機能には Particle Mesh、Particle Mesh Multipole Method、x86 版 Phantom-GRAPE がある。

3.3.1.1 Particle Mesh

Particle Mesh のソースファイルはディレクトリ `src/particle_mesh` の下にある。ここで `Makefile` を編集して、`make` を実行すると、ヘッダファイル `particle_mesh.class.hpp` とライブラリ `libpm.a` ができる。このヘッダファイルをインクルードし、このライブラリをリンクづけすれば、Particle Mesh の機能を使用できるようになる。

3.3.1.2 Particle Mesh Multipole Method

Particle Mesh Multipole Method のソースファイルはディレクトリ `src/particle_mesh_multipole` の下にある。テンプレートライブラリであり、ヘッダファイル `particle_mesh_multipole.hpp` をインクルードすることで使用できる。

3.3.1.3 x86 版 Phantom-GRAPE

x86 版 Phantom-GRAPE のソースファイルはディレクトリ `src/phantom_GRAPE_x86` の下にある。この下には低精度 N 体シミュレーション用、低精度カットオフ付き相互作用計算用、高精度 N 体シミュレーション用 (ディレクトリ `G6/libavx`) がある。それぞれについて述べる。

3.3.1.3.1 低精度 N 体シミュレーション用

これはディレクトリ `src/phantom_GRAPE_x86/G5/newton/libpg5` にある。このディレクトリ内の Makefile を編集して、`make` を実行すると、ライブラリ `libpg5.a` ができる。このディレクトリ内のヘッダファイル `gp5util.h` をインクルードし、ライブラリ `libpg5.a` をリンクすると、この Phantom-GRAPE が使用可能になる。

3.3.1.3.2 低精度カットオフ付き相互作用計算用

これはディレクトリ `src/phantom_GRAPE_x86/G5/table/` にある。このディレクトリ内の Makefile を編集して、`make` を実行すると、ライブラリ `libpg5.a` ができる。このディレクトリ内のヘッダファイル `gp5util.h` をインクルードし、ライブラリ `libpg5.a` をリンクすると、この Phantom-GRAPE が使用可能になる。

3.3.1.3.3 高精度 N 体シミュレーション用

これはディレクトリ `src/phantom_GRAPE_x86/G6/libavx/` にある。このディレクトリ内の Makefile を編集して、`make` を実行すると、ライブラリ `libg6avx.a` ができる。このディレクトリ内のヘッダファイル `gp6util.h` をインクルードし、ライブラリ `libg6avx.a` をリンクすると、この Phantom-GRAPE が使用可能になる。

3.4 テストコード

テストコードはディレクトリ `tests` の下にある。ディレクトリ `tests` にカレントディレクトリを移し、`make check` を実行するとテストスイートが動作する。

3.5 サンプルコード

サンプルコードはディレクトリ `sample` の下にある。サンプルコードは2つ用意されており、重力 N 体シミュレーションと SPH シミュレーションである。

3.5.1 重力 N 体シミュレーション

ディレクトリ `sample/nbody` の下にソースファイルがある。サンプルコードの実行方法はチュートリアルを参照のこと。

3.5.2 SPH シミュレーション

ディレクトリ `sample/sph` の下にソースファイルがある。サンプルコードの実行方法はチュートリアルを参照のこと。

4 コンパイル時のマクロによる選択

4.1 概要

FDPS では、座標系、並列処理の有無、浮動小数点数型の精度、ツリーの最大の深さ、粒子ソートのアルゴリズム等を選択できる。この選択はコンパイル時のマクロの定義によってなされる。以下、それぞれの選択の方法を記述する。

4.2 座標系

4.2.1 概要

座標系は直角座標系 3 次元と直角座標系 2 次元の選択ができる。以下、それらの選択方法について述べる。

4.2.2 直角座標系 3 次元

デフォルトは直角座標系 3 次元である。なにも行わなくても直角座標系 3 次元となる。

4.2.3 直角座標系 2 次元

コンパイル時に `PARTICLE_SIMULATOR_TWO_DIMENSION` をマクロ定義すると直交座標系 2 次元となる。

4.3 並列処理

4.3.1 概要

並列処理に関しては、OpenMP の使用 / 不使用、MPI の使用 / 不使用を選択できる。以下、選択の仕方について記述する。

4.3.2 OpenMP の使用

デフォルトは OpenMP 不使用である。使用する場合は、`PARTICLE_SIMULATOR_THREAD_PARALLEL` をマクロ定義すればよい。GCC コンパイラの場合はコンパイラオプションに `-fopenmp` をつける必要がある。

4.3.3 MPI の使用

デフォルトは MPI 不使用である。使用する場合は、`PARTICLE_SIMULATOR_THREAD_PARALLEL` をマクロ定義すればよい。

4.4 データ型の精度

4.4.1 概要

FDPS 側で用意した Moment クラス (第 7.5.2 節参照) と SuperParticleJ クラス (第 7.6.2 節参照) のデータ型の精度を選択できる。以下、選択の仕方について記述する。

4.4.2 既存の SuperParticleJ クラスと Moment クラスの精度

既存の SuperParticleJ クラスと Moment クラスのメンバ変数の精度はデフォルトで 64 ビットである。32 ビットにしたい場合、
`PARTICLE.SIMULATOR.SPMOM.F32`
をマクロ定義すればよい。

4.5 ツリーの最大深さ (レベル) の変更

ツリー構造は粒子の位置座標のモートンキーを用いて作られているため、ツリーの最大レベルはモートンキーのビット長に依存する。FDPS では 3 次元シミュレーションの場合にのみ、適切なマクロを指定することにより、粒子のモートンキーを 64bit (レベル 21), 96bit (最大レベル 31), 128bit (最大レベル 42) の 3 つから選択することができる。特にマクロを指定しなければ、キーの長さは 128bit になる。キーを 96bit にする場合は `PARTICLE.SIMULATOR.USE_96BIT_KEY` を、64bit にする場合は `PARTICLE.SIMULATOR.USE_64BIT_KEY` をマクロ定義すればよい。

アーキテクチャによっては短いキーとすることで若干の高速化やメモリ節約が実現できるが、特に 64 ビットキーでは粒子分布によってはレベルが不足する。

4.6 粒子のソートの方法の変更

`TreeForForce` クラスの内部では粒子はモートンキーの順でソートされている。デフォルトでは並列ソートアルゴリズムとしてマージソートが使われているが、
`PARTICLE.SIMULATOR.USE_RADIX_SORT`
をマクロ定義することでソートアルゴリズムを基数ソートに、また、
`PARTICLE.SIMULATOR.USE_SAMPLE_SORT`
をマクロ定義することで並列サンプルソートに変更できる。

アーキテクチャによっては基数ソートが若干速いかもしい。また、A64fx では並列サンプルソートが速いことが確認されている。

5 名前空間

5.1 概要

本節では、名前空間の構造について述べる。FDPS は ParticleSimulator という名前空間で囲まれている。以下では、ParticleSimulator 直下にある機能と、ParticleSimulator にネストされている名前空間について述べる。

5.2 ParticleSimulator

FDPS の標準機能すべては名前空間 ParticleSimulator の直下にある。

名前空間 ParticleSimulator は以下のように省略されており、この文書におけるあとの記述でもこの省略形を採用する。

```
namespace PS = ParticleSimulator;
```

名前空間 ParticleSimulator の下にはいくつかの名前空間が拡張機能毎にネストされている。拡張機能には ParticleMesh がある。以下では拡張機能の名前空間について記述する。

5.2.1 ParticleMesh

Particle Mesh の機能は名前空間 ParticleMesh に囲まれており、名前空間 ParticleMesh は名前空間 ParticleSimulator の直下にネストされている。また、ParticleMesh は PM と省略されている。これらをまとめると以下のようにになっている。

```
ParticleSimulator {  
    ParticleMesh {  
    }  
    namespace PM = ParticleMesh;  
}
```

以後、この文書では省略形の PM を用いて記述する。

5.2.2 ParticleMeshMultipoleMethod

Particle Mesh Multipole Method の機能は名前空間 ParticleMeshMultipole に囲まれており、名前空間 ParticleMeshMultipole は名前空間 ParticleSimulator の直下にネストされている。また、ParticleMeshMultipole は PMMM と省略されている。これらをまとめると以下のようにになっている。

```
ParticleSimulator {  
    ParticleMeshMultipole {  
    }  
    namespace PMMM = ParticleMeshMultipole;  
}
```

以後、この文書では省略形の PMMM を用いて記述する。

6 データ型

6.1 概要

FDPS では独自の整数型、実数型、ベクトル型、対称行列型、PS::SEARCH_MODE 型、列挙型が定義されている。整数型、実数型、ベクトル型、対称行列型に関しては必ずしもここに挙げるものを用いる必要はないが、これらを用いることを推奨する。PS::SEARCH_MODE 型、列挙型は必ず用いる必要がある。以下、整数型、実数型、ベクトル型、対称行列型、PS::SEARCH_MODE 型、列挙型の順に記述する。

6.2 整数型

6.2.1 概要

整数型には PS::S32, PS::S64, PS::U32, PS::U64 がある。以下、順にこれらを記述する。

6.2.2 PS::S32

PS::S32 は以下のように定義されている。すなわち 32bit の符号付き整数である。

ソースコード 1: S32

```
1 namespace ParticleSimulator {
2     using S32 = std::int32_t;
3 }
```

6.2.3 PS::S64

PS::S64 は以下のように定義されている。すなわち 64bit の符号付き整数である。

ソースコード 2: S64

```
1 namespace ParticleSimulator {
2     using S64 = std::int64_t;
3 }
```

6.2.4 PS::U32

PS::U32 は以下のように定義されている。すなわち 32bit の符号なし整数である。

ソースコード 3: U32

```
1 namespace ParticleSimulator {
2     using U32 = std::uint32_t;
3 }
```

6.2.5 PS::U64

PS::U64 は以下のように定義されている。すなわち 64bit の符号なし整数である。

ソースコード 4: U64

```
1 namespace ParticleSimulator {
2     using U64 = std::uint64_t;
3 }
```

6.2.6 PS::Count_t

カウント数を表す為の型。現在は PS::U64 型として定義されている。

ソースコード 5: Count_t

```
1 namespace ParticleSimulator {
2     typedef U64 Count_t;
3 }
```

6.3 実数型

6.3.1 概要

実数型には PS::F32, PS::F64 がある。以下、順にこれらを記述する。

6.3.2 PS::F32

PS::F32 は以下のように定義されている。すなわち 32bit の浮動小数点数である。

ソースコード 6: F32

```
1 namespace ParticleSimulator {
2     typedef float F32;
3 }
```

6.3.3 PS::F64

PS::F64 は以下のように定義されている。すなわち 64bit の浮動小数点数である。

ソースコード 7: F64

```
1 namespace ParticleSimulator {
2     typedef double F64;
3 }
```

6.4 ベクトル型

6.4.1 概要

ベクトル型には2次元ベクトル型 `PS::Vector2` と3次元ベクトル型 `PS::Vector3` がある。まずこれら2つを記述する。最後にこれらベクトル型のラッパーについて記述する。

6.4.2 `PS::Vector2`

`PS::Vector2` は x, y の2要素を持つ。これらに対する様々なAPIや演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 8: `Vector2`

```
1 namespace ParticleSimulator{
2     template <typename T>
3     class Vector2{
4     public:
5         //メンバ変数2要素
6         T x, y;
7
8         //コンストラクタ
9         Vector2();
10        Vector2(const T _x, const T _y) : x(_x), y(_y) {}
11        Vector2(const T s) : x(s), y(s) {}
12        Vector2(const Vector2 & src) : x(src.x), y(src.y) {}
13
14        //代入演算子
15        const Vector2 & operator = (const Vector2 & rhs);
16
17        //[]演算子
18        const T & operator[](const int i);
19        T & operator[](const int i);
20
21        //加減算
22        Vector2 operator + (const Vector2 & rhs) const;
23        const Vector2 & operator += (const Vector2 & rhs);
24        Vector2 operator - (const Vector2 & rhs) const;
25        const Vector2 & operator -= (const Vector2 & rhs);
26
27        //ベクトルスカラー積
28        Vector2 operator * (const T s) const;
29        const Vector2 & operator *= (const T s);
```



```

30         friend Vector2 operator * (const T s,
31                                     const Vector2 & v);
32         Vector2 operator / (const T s) const;
33         const Vector2 & operator /= (const T s);
34
35         //内積
36         T operator * (const Vector2 & rhs) const;
37
38         //外積(返り値はスカラ!!)
39         T operator ^ (const Vector2 & rhs) const;
40
41         //Vector2<U>への型変換
42         template <typename U>
43         operator Vector2<U> () const;
44     };
45 }
46 namespace PS = ParticleSimulator;

```

6.4.2.1 コンストラクタ

```

template<typename T>
PS::Vector2<T>::Vector2()

```

- 引数
なし。
- 機能
デフォルトコンストラクタ。メンバ x, y は 0 で初期化される。

```

template<typename T>
PS::Vector2<T>::Vector2(const T _x, const T _y)

```

- 引数
_x: 入力。const T 型。
_y: 入力。const T 型。
- 機能
メンバ x, y をそれぞれ $_x, _y$ で初期化する。

```
template<typename T>
PS::Vector2<T>::Vector2(const T s);
```

- 引数
s: 入力。const T 型。
- 機能
メンバ x 、 y を両方とも s の値で初期化する。

6.4.2.2 コピーコンストラクタ

```
template<typename T>
PS::Vector2<T>::Vector2(const PS::Vector2<T> & src)
```

- 引数
src: 入力。const PS::Vector2<T> &型。
- 機能
コピーコンストラクタ。src で初期化する。

6.4.2.3 メンバ変数

```
template<typename T>
T PS::Vector2<T>::x;

template<typename T>
T PS::Vector2<T>::y;
```

- 機能
メンバ x 、 y を直接操作出来る。

6.4.2.4 代入演算子

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator =
    (const PS::Vector2<T> & rhs);
```

- 引数

rhs: 入力。const PS::Vector2<T> &型。

- 返回值

const PS::Vector2<T> &型。rhs の x,y の値を自身のメンバ x,y に代入し自身の参照を返す。代入演算子。

6.4.2.5 [] 演算子

```
template<typename T>
const T & PS::Vector2<T>::operator[]
    (const int i);
```

- 引数

i: 入力。const int 型。

- 返回值

const <T> &型。ベクトルの i 成分を返す。

- 備考

直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

```
template<typename T>
T & PS::Vector2<T>::operator[]
    (const int i);
```

- 引数

i: 入力。const int 型。

- 返回值

<T> &型。ベクトルの i 成分を返す。

- 備考

直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

6.4.2.6 加減算

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator +
    (const PS::Vector2<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::Vector2<T> &型。

- 返り値

PS::Vector2<T> 型。rhs の x,y の値と自身のメンバ x,y の値の和を取った値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator +=
    (const PS::Vector2<T> & rhs);
```

- 引数

rhs: 入力。const PS::Vector2<T> &型。

- 返り値

const PS::Vector2<T> &型。rhs の x,y の値を自身のメンバ x,y に足し、自身を返す。

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator -
    (const PS::Vector2<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::Vector2<T> &型。

- 返り値

PS::Vector2<T> 型。rhs の x,y の値と自身のメンバ x,y の値の差を取った値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator -=
    (const PS::Vector2<T> & rhs);
```

- 引数

rhs: 入力。const PS::Vector2<T> &型。

- 返り値

const PS::Vector2<T> &型。自身のメンバ x,y から rhs の x,y を引き自身を返す。

6.4.2.7 ベクトルスカラ積

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator * (const T s) const;
```

- 引数

s: 入力。const T 型。

- 返回值

PS::Vector2<T> 型。自身のメンバ x, y それぞれに s をかけた値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator *= (const T s);
```

- 引数

rhs: 入力。const T 型。

- 返回值

const PS::Vector2<T> &型。自身のメンバ x, y それぞれに s をかけ自身を返す。

```
template<typename T>
PS::Vector2<T> PS::Vector2<T>::operator / (const T s) const;
```

- 引数

s: 入力。const T 型。

- 返回值

PS::Vector2<T> 型。自身のメンバ x, y それぞれを s で割った値を返す。

```
template<typename T>
const PS::Vector2<T> & PS::Vector2<T>::operator /= (const T s);
```

- 引数

rhs: 入力。const T 型。

- 返回值

const PS::Vector2<T> &型。自身のメンバ x, y それぞれを s で割り自身を返す。

6.4.2.8 内積、外積

```
template<typename T>
T PS::Vector2<T>::operator * (const PS::Vector2<T> & rhs) const;
```

- 引数
rhs: 入力。const PS::Vector2<T> &型。
- 返回值
T 型。自身と rhs の内積を取った値を返す。

```
template<typename T>
T PS::Vector2<T>::operator ^ (const PS::Vector2<T> & rhs) const;
```

- 引数
rhs: 入力。const PS::Vector2<T> &型。
- 返回值
T 型。自身と rhs の外積を取った値を返す。

6.4.2.9 Vector2<U> への型変換

```
template<typename T>
template <typename U>
PS::Vector2<T>::operator PS::Vector2<U> () const;
```

- 引数
なし。
- 返回值
const PS::Vector2<U> 型。
- 機能
const PS::Vector2<T> 型を const PS::Vector2<U> 型にキャストする。

6.4.3 PS::Vector3

PS::Vecotr3 は x, y, z の 3 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 9: Vector3

```
1 namespace ParticleSimulator{
2     template <typename T>
3     class Vector3{
4     public:
5         //メンバ変数は以下の二つのみ。
6         T x, y, z;
7
8         //コンストラクタ
9         Vector3() : x(T(0)), y(T(0)), z(T(0)) {}
10        Vector3(const T _x, const T _y, const T _z) : x(_x), y(
            _y), z(_z) {}
11        Vector3(const T s) : x(s), y(s), z(s) {}
12        Vector3(const Vector3 & src) : x(src.x), y(src.y), z(
            src.z) {}
13
14        //代入演算子
15        const Vector3 & operator = (const Vector3 & rhs);
16
17        //[] 演算子
18        const T & opertor[](const int i);
19        T & operator[](const int i);
20
21        //加減算
22        Vector3 operator + (const Vector3 & rhs) const;
23        const Vector3 & operator += (const Vector3 & rhs);
24        Vector3 operator - (const Vector3 & rhs) const;
25        const Vector3 & operator -= (const Vector3 & rhs);
26
27        //ベクトルスカラ積
28        Vector3 operator * (const T s) const;
29        const Vector3 & operator *= (const T s);
30        friend Vector3 operator * (const T s, const Vector3 & v
            );
31        Vector3 operator / (const T s) const;
32        const Vector3 & operator /= (const T s);
```

```

33
34     //内積
35     T operator * (const Vector3 & rhs) const;
36
37     //外積(返り値はスカラー!!)
38     T operator ^ (const Vector3 & rhs) const;
39
40     //Vector3<U>への型変換
41     template <typename U>
42     operator Vector3<U> () const;
43 };
44 }

```

6.4.3.1 コンストラクタ

```

template<typename T>
PS::Vector3<T>::Vector3()

```

- 引数
なし。
- 機能
デフォルトコンストラクタ。メンバ x, y は 0 で初期化される。

```

template<typename T>
PS::Vector3<T>::Vector3(const T _x, const T _y)

```

- 引数
_x: 入力。const T 型。
_y: 入力。const T 型。
- 機能
メンバ x, y をそれぞれ $_x, _y$ で初期化する。

```

template<typename T>
PS::Vector3<T>::Vector3(const T s);

```


- 引数

s: 入力。const T 型。

- 機能

メンバ x 、 y を両方とも s の値で初期化する。

6.4.3.2 コピーコンストラクタ

```
template<typename T>
PS::Vector3<T>::Vector3(const PS::Vector3<T> & src)
```

- 引数

src: 入力。const PS::Vector3<T> &型。

- 機能

コピーコンストラクタ。src で初期化する。

6.4.3.3 メンバ変数

```
template<typename T>
T PS::Vector3<T>::x;

template<typename T>
T PS::Vector3<T>::y;

template<typename T>
T PS::Vector3<T>::z;
```

- 機能

メンバ x 、 y 、 z を直接操作出来る。

6.4.3.4 代入演算子

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator =
    (const PS::Vector3<T> & rhs);
```

- 引数

rhs: 入力。const PS::Vector3<T> &型。

- 返回值

const PS::Vector3<T> &型。rhs の x,y の値を自身のメンバ x,y に代入し自身の参照を返す。代入演算子。

6.4.3.5 [] 演算子

```
template<typename T>
const T & PS::Vector3<T>::operator[]
    (const int i);
```

- 引数

i: 入力。const int 型。

- 返回值

const <T> &型。ベクトルの i 成分を返す。

- 備考

直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

```
template<typename T>
T & PS::Vector3<T>::operator[]
    (const int i);
```

- 引数

i: 入力。const int 型。

- 返回值

<T> &型。ベクトルの i 成分を返す。

- 備考

直接メンバ変数を指定する場合に比べ、処理が遅くなることがある。

6.4.3.6 加減算

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator +
    (const PS::Vector3<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::Vector3<T> &型。

- 返り値

PS::Vector3<T> 型。rhs の x,y の値と自身のメンバ x,y の値の和を取った値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator +=
    (const PS::Vector3<T> & rhs);
```

- 引数

rhs: 入力。const PS::Vector3<T> &型。

- 返り値

const PS::Vector3<T> &型。rhs の x,y の値を自身のメンバ x,y に足し、自身を返す。

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator -
    (const PS::Vector3<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::Vector3<T> &型。

- 返り値

PS::Vector3<T> 型。rhs の x,y の値と自身のメンバ x,y の値の差を取った値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator -=
    (const PS::Vector3<T> & rhs);
```

- 引数

rhs: 入力。const PS::Vector3<T> &型。

- 返り値

const PS::Vector3<T> &型。自身のメンバ x,y から rhs の x,y を引き自身を返す。

6.4.3.7 ベクトルスカラ積

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator * (const T s) const;
```

- 引数

s: 入力。const T 型。

- 返り値

PS::Vector3<T> 型。自身のメンバ x,y それぞれに s をかけた値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator *= (const T s);
```

- 引数

rhs: 入力。const T 型。

- 返り値

const PS::Vector3<T> &型。自身のメンバ x,y それぞれに s をかけ自身を返す。

```
template<typename T>
PS::Vector3<T> PS::Vector3<T>::operator / (const T s) const;
```

- 引数

s: 入力。const T 型。

- 返り値

PS::Vector3<T> 型。自身のメンバ x,y それぞれを s で割った値を返す。

```
template<typename T>
const PS::Vector3<T> & PS::Vector3<T>::operator /= (const T s);
```

- 引数
rhs: 入力。const T 型。
- 返回值
const PS::Vector3<T> &型。自身のメンバ x, y それぞれを s で割り自身を返す。

6.4.3.8 内積、外積

```
template<typename T>
T PS::Vector3<T>::operator * (const PS::Vector3<T> & rhs) const;
```

- 引数
rhs: 入力。const PS::Vector3<T> &型。
- 返回值
T 型。自身と rhs の内積を取った値を返す。

```
template<typename T>
T PS::Vector3<T>::operator ^ (const PS::Vector3<T> & rhs) const;
```

- 引数
rhs: 入力。const PS::Vector3<T> &型。
- 返回值
T 型。自身と rhs の外積を取った値を返す。

6.4.3.9 Vector3<U> への型変換

```
template<typename T>
template <typename U>
PS::Vector3<T>::operator PS::Vector3<U> () const;
```

- 引数

なし

- 返回值

const PS::Vector3<U> 型。

- 機能

const PS::Vector3<T> 型を const PS::Vector3<U> 型にキャストする。

6.4.4 ベクトル型のラッパー

ベクトル型のラッパーの定義を以下に示す。

ソースコード 10: vectorwrapper

```
1 namespace ParticleSimulator{
2     typedef Vector2<F32> F32vec2;
3     typedef Vector3<F32> F32vec3;
4     typedef Vector2<F64> F64vec2;
5     typedef Vector3<F64> F64vec3;
6 #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
7     typedef F32vec2 F32vec;
8     typedef F64vec2 F64vec;
9 #else
10    typedef F32vec3 F32vec;
11    typedef F64vec3 F64vec;
12 #endif
13 }
```

すなわち PS::F32vec2, PS::F32vec3, PS::F64vec2, PS::F64vec3 はそれぞれ単精度 2 次元ベクトル、倍精度 2 次元ベクトル、単精度 3 次元ベクトル、倍精度 3 次元ベクトルである。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32vec と PS::F64vec はそれぞれ単精度 2 次元ベクトル、倍精度 2 次元ベクトルとなる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32vec と PS::F64vec はそれぞれ単精度 3 次元ベクトル、倍精度 3 次元ベクトルとなる。

6.5 オルソトープ型

6.5.1 概要

オルソトープ型 (長方形 [空間 2 次元] や直方体 [空間 3 次元] を表すデータ型) には 2 次元オルソトープ型 PS::Orthotope2 と 3 次元オルソトープ型 PS::Orthotope3 がある。まずこれから 2 つを記述する。最後にこれらオルソトープ型のラッパーについて記述する。

6.5.2 PS::Orthotope2

PS::Orthotope2 は PS::Vector2 型のメンバ変数 `low_`, `high_` を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 11: Orthotope2

```
1 namespace ParticleSimulator{
2     template<class T>
3     class Orthotope2{
4     public:
5         Vector2<T> low_;
6         Vector2<T> high_;
7
8         Orthotope2(): low_(9999.9), high_(-9999.9){}
9
10        Orthotope2(const Vector2<T> & _low, const Vector2<T> &
11                    _high)
12            : low_(low_), high_(high_){}
13
14        Orthotope2(const Orthotope2 & src) : low_(src.low_),
15            high_(src.high_){}
16
17        Orthotope2(const Vector2<T> & center, const T length) :
18            low_(center-(Vector2<T>)(length)), high_(center+(
19                Vector2<T>)(length)) {
20
21        }
22
23        void initNegativeVolume(){
24            low_ = std::numeric_limits<float>::max() / 128;
25            high_ = -low_;
26        }
27
28        void init(){
29            initNegativeVolume();
30        }
31
32        void merge( const Orthotope2 & ort ){
33            this->high_.x = ( this->high_.x > ort.high_.x ) ?
34                this->high_.x : ort.high_.x;
35            this->high_.y = ( this->high_.y > ort.high_.y ) ?
```

```

        this->high_.y : ort.high_.y;
31     this->low_.x = ( this->low_.x <= ort.low_.x ) ?
        this->low_.x : ort.low_.x;
32     this->low_.y = ( this->low_.y <= ort.low_.y ) ?
        this->low_.y : ort.low_.y;
33 }
34
35 void merge( const Vector2<T> & vec ){
36     this->high_.x = ( this->high_.x > vec.x ) ? this->
        high_.x : vec.x;
37     this->high_.y = ( this->high_.y > vec.y ) ? this->
        high_.y : vec.y;
38     this->low_.x = ( this->low_.x <= vec.x ) ? this->
        low_.x : vec.x;
39     this->low_.y = ( this->low_.y <= vec.y ) ? this->
        low_.y : vec.y;
40 }
41
42 void merge( const Vector2<T> & vec, const T size){
43     this->high_.x = ( this->high_.x > vec.x + size ) ?
        this->high_.x : vec.x + size;
44     this->high_.y = ( this->high_.y > vec.y + size ) ?
        this->high_.y : vec.y + size;
45     this->low_.x = ( this->low_.x <= vec.x - size ) ?
        this->low_.x : vec.x - size;
46     this->low_.y = ( this->low_.y <= vec.y - size ) ?
        this->low_.y : vec.y - size;
47 }
48 };
49 }
50 namespace PS = ParticleSimulator;

```

6.5.2.1 メンバ変数

```
template<typename T>
PS::Vector2<T> PS::Orthotope2<T>::low_;

template<typename T>
PS::Vector2<T> PS::Orthotope2<T>::high_;
```

- 機能

メンバ変数 `low_`、`high_` を直接操作出来る。

6.5.2.2 コンストラクタ

```
template<typename T>
PS::Orthotope2<T>::Orthotope2();
```

- 引数

なし。ただし、テンプレート引数 `T` は `PS::F32` または `PS::F64` でなければならない。

- 機能

デフォルトコンストラクタ。メンバ変数 `low_`、`high_` は、それぞれ、`(9999.9, 9999.9)`、`(-9999.9, -9999.9)` で初期化される。

```
template<typename T>
PS::Orthotope2<T>::Orthotope2(const Vector2<T> _low, const Vector2<T> _high);
```

- 引数

`_low`: 入力。const Vector2<T> 型。

`_high`: 入力。const Vector2<T> 型。

ここで、`T` は `PS::F32` または `PS::F64` でなければならない。

- 機能

メンバ変数 `low_`、`high_` を、それぞれ `_low`、`_high` で初期化する。

```
template<typename T>
PS::Orthotope2<T>::Orthotope2(const Vector2<T> & center, const T length);
```

- 引数

center: 入力。const Vector2<T> &

length: 入力。const T 型。

- 機能

メンバ変数 low_、high_ を、それぞれ、center-(Vector2<T>)(length)、center+(Vector2<T>)(length) で初期化する。

6.5.2.3 コピーコンストラクタ

```
template<typename T>
```

```
PS::Orthotope2<T>::Orthotope2(const Orthotope2<T> & src);
```

- 引数

src: 入力。const Orthotope2<T> &型。

- 機能

メンバ変数 low_、high_ を、それぞれ、src.low_、src.high_ で初期化する。

6.5.2.4 初期化

```
template<typename T>
```

```
PS::Orthotope2<T>::initNegativeVolume();
```

- 引数

なし。

- 機能

メンバ変数 low_、high_ を、それぞれ、 (a, a) 、 $(-a, -a)$ で初期化する。ここで、 $a = \text{std::numeric_limits}<T>::max() / 128$ である。

```
template<typename T>
```

```
PS::Orthotope2<T>::init();
```

- 引数

なし。

- 機能

メンバ関数 initNegativeVolume と同じ機能を提供する。

6.5.2.5 結合操作

```
template<typename T>
void PS::Orthotope2<T>::merge( const Orthotope2 & ort )();
```

- 引数

ort: 入力。const Orthotope2 &型。

- 機能

メンバ変数 low_、high_を、当該長方形と長方形 ort を包含する最小の長方形を記述するように更新する。

```
template<typename T>
void PS::Orthotope2<T>::merge( const Vector2<T> & vec )();
```

- 引数

vec: 入力。const Vector2<T> &型。

- 機能

メンバ変数 low_、high_を、点 vec を包含するように更新する。

```
template<typename T>
void PS::Orthotope2<T>::merge( const Vector2<T> & vec, const T size )();
```

- 引数

vec: 入力。const Vector2<T> &型。

size: 入力。const T 型。

- 機能

メンバ変数 low_、high_を、中心 vec、半径 size の円を包含するように更新する。

6.5.3 PS::Orthotope3

PS::Orthotope3 は PS::Vector3 型のメンバ変数 low_、high_を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 12: Orthotope3

```
1 namespace ParticleSimulator{
2     template<class T>
3     class Orthotope3{
4     public:
5         Vector3<T> low_;
6         Vector3<T> high_;
7
8         Orthotope3(): low_(9999.9), high_(-9999.9){}
9
10        Orthotope3(const Vector3<T> & _low, const Vector3<T> &
11                _high)
12            : low_( _low), high_( _high) {}
13
14        Orthotope3(const Orthotope3 & src) : low_(src.low_),
15                high_(src.high_){}
16
17        Orthotope3(const Vector3<T> & center, const T length) :
18            low_(center-(Vector3<T>)(length)), high_(center+(
19                Vector3<T>)(length)) {
20
21        }
22
23        void initNegativeVolume(){
24            low_ = std::numeric_limits<float>::max() / 128;
25            high_ = -low_;
26        }
27
28        void init(){
29            initNegativeVolume();
30        }
31
32        void merge( const Orthotope3 & ort ){
33            this->high_.x = ( this->high_.x > ort.high_.x ) ?
34                this->high_.x : ort.high_.x;
35            this->high_.y = ( this->high_.y > ort.high_.y ) ?
36                this->high_.y : ort.high_.y;
37            this->high_.z = ( this->high_.z > ort.high_.z ) ?
38                this->high_.z : ort.high_.z;
39            this->low_.x = ( this->low_.x <= ort.low_.x ) ?
40                this->low_.x : ort.low_.x;
```

```

33         this->low_.y = ( this->low_.y <= ort.low_.y ) ?
           this->low_.y : ort.low_.y;
34         this->low_.z = ( this->low_.z <= ort.low_.z ) ?
           this->low_.z : ort.low_.z;
35     }
36
37     void merge( const Vector3<T> & vec ){
38         this->high_.x = ( this->high_.x > vec.x ) ? this->
           high_.x : vec.x;
39         this->high_.y = ( this->high_.y > vec.y ) ? this->
           high_.y : vec.y;
40         this->high_.z = ( this->high_.z > vec.z ) ? this->
           high_.z : vec.z;
41         this->low_.x = ( this->low_.x <= vec.x ) ? this->
           low_.x : vec.x;
42         this->low_.y = ( this->low_.y <= vec.y ) ? this->
           low_.y : vec.y;
43         this->low_.z = ( this->low_.z <= vec.z ) ? this->
           low_.z : vec.z;
44     }
45
46     void merge( const Vector3<T> & vec, const T size){
47         this->high_.x = ( this->high_.x > vec.x + size ) ?
           this->high_.x : vec.x + size;
48         this->high_.y = ( this->high_.y > vec.y + size ) ?
           this->high_.y : vec.y + size;
49         this->high_.z = ( this->high_.z > vec.z + size ) ?
           this->high_.z : vec.z + size;
50         this->low_.x = ( this->low_.x <= vec.x - size ) ?
           this->low_.x : vec.x - size;
51         this->low_.y = ( this->low_.y <= vec.y - size ) ?
           this->low_.y : vec.y - size;
52         this->low_.z = ( this->low_.z <= vec.z - size ) ?
           this->low_.z : vec.z - size;
53     }
54 };
55 }
56 namespace PS = ParticleSimulator;

```

6.5.3.1 メンバ変数

```
template<typename T>
PS::Vector3<T> PS::Orthotope3<T>::low_;

template<typename T>
PS::Vector3<T> PS::Orthotope3<T>::high_;
```

- 機能

メンバ変数 `low_`、`high_` を直接操作出来る。

6.5.3.2 コンストラクタ

```
template<typename T>
PS::Orthotope3<T>::Orthotope3();
```

- 引数

なし。ただし、テンプレート引数 `T` は `PS::F32` または `PS::F64` でなければならない。

- 機能

デフォルトコンストラクタ。メンバ変数 `low_`、`high_` は、それぞれ、`(9999.9, 9999.9, 9999.9)`、`(-9999.9, -9999.9, -9999.9)` で初期化される。

```
template<typename T>
PS::Orthotope3<T>::Orthotope3(const Vector3<T> _low, const Vector3<T> _high);
```

- 引数

`_low`: 入力。const Vector3<T> 型。

`_high`: 入力。const Vector3<T> 型。

ここで、`T` は `PS::F32` または `PS::F64` でなければならない。

- 機能

メンバ変数 `low_`、`high_` を、それぞれ `_low`、`_high` で初期化する。

```
template<typename T>
PS::Orthotope3<T>::Orthotope3(const Vector3<T> & center, const T length);
```

- 引数

center: 入力。const Vector3<T> &

length: 入力。const T 型。

- 機能

メンバ変数 low_、high_ を、それぞれ、center-(Vector3<T>)(length)、center+(Vector3<T>)(length) で初期化する。

6.5.3.3 コピーコンストラクタ

```
template<typename T>
PS::Orthotope3<T>::Orthotope3(const Orthotope3<T> & src);
```

- 引数

src: 入力。const Orthotope3<T> &型。

- 機能

メンバ変数 low_、high_ を、それぞれ、src.low_、src.high_ で初期化する。

6.5.3.4 初期化

```
template<typename T>
PS::Orthotope3<T>::initNegativeVolume();
```

- 引数

なし。

- 機能

メンバ変数 low_、high_ を、それぞれ、 (a, a, a) 、 $(-a, -a, -a)$ で初期化する。ここで、 $a = \text{std::numeric_limits}<\text{float}>::\text{max}() / 128$ である。

```
template<typename T>
PS::Orthotope3<T>::init();
```

- 引数

なし。

- 機能

メンバ関数 initNegativeVolume と同じ機能を提供する。

6.5.3.5 結合操作

```
template<typename T>
void PS::Orthotope3<T>::merge( const Orthotope3 & ort )();
```

- 引数

ort: 入力。const Orthotope3 &型。

- 機能

メンバ変数 low_、high_を、当該長方形と長方形 ort を包含する最小の長方形を記述するように更新する。

```
template<typename T>
void PS::Orthotope3<T>::merge( const Vector3<T> & vec )();
```

- 引数

vec: 入力。const Vector3<T> &型。

- 機能

メンバ変数 low_、high_を、点 vec を包含するように更新する。

```
template<typename T>
void PS::Orthotope3<T>::merge( const Vector3<T> & vec, const T size )();
```

- 引数

vec: 入力。const Vector3<T> &型。

size: 入力。const T 型。

- 機能

メンバ変数 low_、high_を、中心 vec、半径 size の球を包含するように更新する。

6.5.4 オルソトープ型のラッパー

オルソトープ型のラッパーの定義を以下に示す。

ソースコード 13: Orthotope Wrapper

```
1 namespace ParticleSimulator{
2     typedef Orthotope2<F32> F32ort2;
3     typedef Orthotope3<F32> F32ort3;
4     typedef Orthotope2<F64> F64ort2;
5     typedef Orthotope3<F64> F64ort3;
6 #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
7     typedef F32ort2 F32ort;
8     typedef F64ort2 F64ort;
9 #else
10    typedef F32ort3 F32ort;
11    typedef F64ort3 F64ort;
12 #endif
13 }
```

すなわち PS::F32ort2, PS::F32ort3, PS::F64ort2, PS::F64ort3 はそれぞれ単精度 2 次元オルソトープ、倍精度 2 次元オルソトープ、単精度 3 次元オルソトープ、倍精度 3 次元オルソトープである。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32ort と PS::F64ort はそれぞれ単精度 2 次元オルソトープ、倍精度 2 次元オルソトープとなる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32ort と PS::F64ort はそれぞれ単精度 3 次元オルソトープ、倍精度 3 次元オルソトープとなる。

6.6 対称行列型

6.6.1 概要

対称行列型には 2x2 対称行列型 PS::MatrixSym2 と 3x3 対称行列型 PS::MatrixSym3 がある。まずこれら 2 つを記述する。最後にこれら対称行列型のラッパーについて記述する。

6.6.2 PS::MatrixSym2

PS::MatrixSym2 は xx, yy, xy の 3 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 14: MatrixSym2

```
1 namespace ParticleSimulator{
2     template<class T>
3     class MatrixSym2{
4     public:
5         // メンバ変数 3 要素
6         T xx, yy, xy;
```

```

7
8      // コンストラクタ
9      MatrixSym2() : xx(T(0)), yy(T(0)), xy(T(0)) {}
10     MatrixSym2(const T _xx, const T _yy, const T _xy)
11         : xx(_xx), yy(_yy), xy(_xy) {}
12     MatrixSym2(const T s) : xx(s), yy(s), xy(s){}
13     MatrixSym2(const MatrixSym2 & src) : xx(src.xx), yy(src
        .yy), xy(src.xy) {}
14
15     // 代入演算子
16     const MatrixSym2 & operator = (const MatrixSym2 & rhs);
17
18     // 加減算
19     MatrixSym2 operator + (const MatrixSym2 & rhs) const;
20     const MatrixSym2 & operator += (const MatrixSym2 & rhs)
        const;
21     MatrixSym2 operator - (const MatrixSym2 & rhs) const;
22     const MatrixSym2 & operator -= (const MatrixSym2 & rhs)
        const;
23
24     // トレースの計算
25     T getTrace() const;
26
27     // MatrixSym2<U>への型変換
28     template <typename U>
29     operator MatrixSym2<U> () const;
30 }
31 }
32 namespace PS = ParticleSimulator;

```

6.6.2.1 コンストラクタ

```

template<typename T>
PS::MatrixSym2<T>::MatrixSym2();

```

- 引数
なし。
- 機能

デフォルトコンストラクタ。メンバ `xx,yy,xy` は 0 で初期化される。

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2
    (const T _xx,
     const T _yy,
     const T _xy);
```

- 引数

`_xx`: 入力。const T 型。

`_yy`: 入力。const T 型。

`_xy`: 入力。const T 型。

- 機能

メンバ `xx`、`yy`、`xy` をそれぞれ `_xx`、`_yy`、`_xy` で初期化する。

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2(const T s);
```

- 引数

`s`: 入力。const T 型。

- 機能

メンバ `xx`、`yy`、`xy` すべてを `s` の値で初期化する。

6.6.2.2 コピーコンストラクタ

```
template<typename T>
PS::MatrixSym2<T>::MatrixSym2(const PS::MatrixSym2<T> & src)
```

- 引数

`src`: 入力。const PS::MatrixSym2<T> &型。

- 機能

コピーコンストラクタ。`src` で初期化する。

6.6.2.3 代入演算子

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator =
    (const PS::MatrixSym2<T> & rhs);
```

- 引数

rhs: 入力。const PS::MatrixSym2<T> &型。

- 返回值

const PS::MatrixSym2<T> &型。rhs の xx,yy,xy の値を自身のメンバ xx,yy,xy に代入し自身の参照を返す。代入演算子。

6.6.2.4 加減算

```
template<typename T>
PS::MatrixSym2<T> PS::MatrixSym2<T>::operator +
    (const PS::MatrixSym2<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::MatrixSym2<T> &型。

- 返回值

PS::MatrixSym2<T> 型。rhs の xx,yy,xy の値と自身のメンバ xx,yy,xy の値の和を取った値を返す。

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator +=
    (const PS::MatrixSym2<T> & rhs);
```

- 引数

rhs: 入力。const PS::MatrixSym2<T> &型。

- 返回值

const PS::MatrixSym2<T> &型。rhs の xx,yy,xy の値を自身のメンバ xx,yy,xy に足し、自身を返す。

```
template<typename T>
PS::MatrixSym2<T> PS::MatrixSym2<T>::operator -
    (const PS::MatrixSym2<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::MatrixSym2<T> &型。

- 返回值

PS::MatrixSym2<T> 型。rhs の xx,yy,xy の値と自身のメンバ xx,yy,xy の値の差を取った値を返す。

```
template<typename T>
const PS::MatrixSym2<T> & PS::MatrixSym2<T>::operator -=
    (const PS::MatrixSym2<T> & rhs);
```

- 引数

rhs: 入力。const PS::MatrixSym2<T> &型。

- 返回值

const PS::MatrixSym2<T> &型。自身のメンバ xx,yy,xy から rhs の xx,yy,xy を引き自身を返す。

6.6.2.5 トレースの計算

```
template<typename T>
T PS::MatrixSym2<T>::getTrace() const;
```

- 引数

なし

- 返回值

T 型。

- 機能

トレースを計算し、その結果を返す。

6.6.2.6 MatrixSym2<U> への型変換

```
template<typename T>
template<typename U>
PS::MatrixSym2<T>::operator PS::MatrixSym2<U> () const;
```

- 引数
なし。
- 返回值
const PS::MatrixSym2<U> 型。
- 機能
const PS::MatrixSym2<T> 型を const PS::MatrixSym2<U> 型にキャストする

6.6.3 PS::MatrixSym3

PS::MatrixSym3 は xx, yy, zz, xy, xz, yz の 6 要素を持つ。これらに対する様々な API や演算子を定義した。それらの宣言を以下に記述する。この節ではこれらについて詳しく記述する。

ソースコード 15: MatrixSym3

```
1 namespace ParticleSimulator{
2     template<class T>
3     class MatrixSym3{
4     public:
5         // メンバ変数 6 要素
6         T xx, yy, zz, xy, xz, yz;
7
8         // コンストラクタ
9         MatrixSym3() : xx(T(0)), yy(T(0)), zz(T(0)),
10                        xy(T(0)), xz(T(0)), yz(T(0)) {}
11         MatrixSym3(const T _xx, const T _yy, const T _zz,
12                    const T _xy, const T _xz, const T _yz )
13             : xx(_xx), yy(_yy), zz(_zz),
14               xy(_xy), xz(_xz), yz(_yz) {}
15         MatrixSym3(const T s) : xx(s), yy(s), zz(s),
16                                xy(s), xz(s), yz(s) {}
17         MatrixSym3(const MatrixSym3 & src) :
18             xx(src.xx), yy(src.yy), zz(src.zz),
```

```

19         xy(src.xy), xz(src.xz), yz(src.yz) {}
20
21     // 代入演算子
22     const MatrixSym3 & operator = (const MatrixSym3 & rhs);
23
24     // 加減算
25     MatrixSym3 operator + (const MatrixSym3 & rhs) const;
26     const MatrixSym3 & operator += (const MatrixSym3 & rhs)
27         const;
28     MatrixSym3 operator - (const MatrixSym3 & rhs) const;
29     const MatrixSym3 & operator -= (const MatrixSym3 & rhs)
30         const;
31
32     // トレースを取る
33     T getTrace() const;
34
35     // MatrixSym3<U>への型変換
36     template <typename U>
37     operator MatrixSym3<U> () const;
38 }
39 namespace PS = ParticleSimulator;

```

6.6.3.1 コンストラクタ

```

template<typename T>
PS::MatrixSym3<T>::MatrixSym3();

```

- 引数

なし。

- 機能

デフォルトコンストラクタ。6要素は0で初期化される。

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const T _xx,
                               const T _yy,
                               const T _zz,
                               const T _xy,
                               const T _xz,
                               const T _yz);
```

- 引数

_xx: 入力。const T 型。

_yy: 入力。const T 型。

_zz: 入力。const T 型。

_xy: 入力。const T 型。

_xz: 入力。const T 型。

_yz: 入力。const T 型。

- 機能

メンバ xx、yy、zz、xy、xz、yz をそれぞれ _xx、_yy、_zz、_xy、_xz、_yz で初期化する。

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const T s);
```

- 引数

s: 入力。const T 型。

- 機能

6 要素すべてを s の値で初期化する。

6.6.3.2 コピーコンストラクタ

```
template<typename T>
PS::MatrixSym3<T>::MatrixSym3(const PS::MatrixSym3<T> & src)
```

- 引数

src: 入力。const PS::MatrixSym3<T> &型。

- 機能

コピーコンストラクタ。src で初期化する。

6.6.3.3 代入演算子

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator =
    (const PS::MatrixSym3<T> & rhs);
```

- 引数

rhs: 入力。const PS::MatrixSym3<T> &型。

- 返り値

const PS::MatrixSym3<T> &型。rhs の 6 要素それぞれの値を自身の 6 要素それぞれに代入し自身の参照を返す。代入演算子。

6.6.3.4 加減算

```
template<typename T>
PS::MatrixSym3<T> PS::MatrixSym3<T>::operator +
    (const PS::MatrixSym3<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::MatrixSym3<T> &型。

- 返り値

PS::MatrixSym3<T> 型。rhs の 6 要素それぞれの値と自身の 6 要素の値の和を取った値を返す。

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator +=
    (const PS::MatrixSym3<T> & rhs);
```

- 引数

rhs: 入力。const PS::MatrixSym3<T> &型。

- 返り値

const PS::MatrixSym3<T> &型。rhs の 6 要素それぞれの値を自身の 6 要素それぞれに足し、自身を返す。

```
template<typename T>
PS::MatrixSym3<T> PS::MatrixSym3<T>::operator -
    (const PS::MatrixSym3<T> & rhs) const;
```

- 引数

rhs: 入力。const PS::MatrixSym3<T> &型。

- 返回值

PS::MatrixSym3<T> 型。rhs の 6 要素それぞれの値と自身の 6 要素それぞれの値の差を取った値を返す。

```
template<typename T>
const PS::MatrixSym3<T> & PS::MatrixSym3<T>::operator -=
    (const PS::MatrixSym3<T> & rhs);
```

- 引数

rhs: 入力。const PS::MatrixSym3<T> &型。

- 返回值

const PS::MatrixSym3<T> &型。自身の 6 要素それぞれから rhs の 6 要素それぞれを引き自身を返す。

6.6.3.5 トレースの計算

```
template<typename T>
T PS::MatrixSym3<T>::getTrace() const;
```

- 引数

なし

- 返回值

T 型。

- 機能

トレースを計算し、その結果を返す。

6.6.3.6 MatrixSym3<U> への型変換

```
template<typename T>
template<typename U>
PS::MatrixSym3<T>::operator PS::MatrixSym3<U> () const;
```

- 引数
なし。
- 返回值
const PS::MatrixSym3<U> 型。
- 機能
const PS::MatrixSym3<T> 型を const PS::MatrixSym3<U> 型にキャストする

6.6.4 対称行列型のラッパー

対称行列型のラッパーの定義を以下に示す。

ソースコード 16: matrixsymwrapper

```
1 namespace ParticleSimulator{
2     typedef MatrixSym2<F32> F32mat2;
3     typedef MatrixSym3<F32> F32mat3;
4     typedef MatrixSym2<F64> F64mat2;
5     typedef MatrixSym3<F64> F64mat3;
6 #ifdef PARTICLE_SIMULATOR_TWO_DIMENSION
7     typedef F32mat2 F32mat;
8     typedef F64mat2 F64mat;
9 #else
10    typedef F32mat3 F32mat;
11    typedef F64mat3 F64mat;
12 #endif
13 }
14 namespace PS = ParticleSimulator;
```

すなわち PS::F32mat2, PS::F32mat3, PS::F64mat2, PS::F64mat3 はそれぞれ単精度 2x2 対称行列、倍精度 2x2 対称行列、単精度 3x3 対称行列、倍精度 3x3 対称行列である。FDPS で扱う空間座標系を 2 次元とした場合、PS::F32mat と PS::F64mat はそれぞれ単精度 2x2 対称行列、倍精度 2x2 対称行列となる。一方、FDPS で扱う空間座標系を 3 次元とした場合、PS::F32mat と PS::F64mat はそれぞれ単精度 3x3 対称行列、倍精度 3x3 対称行列となる。

6.7 PS::SEARCH_MODE 型

6.7.1 概要

本節では、PS::SEARCH_MODE 型について記述する。PS::SEARCH_MODE 型は相互作用ツリークラスのテンプレート引数としてのみ使用されるものである。この型によって、相互作用ツリークラスで計算する相互作用のモードを決定する。PS::SEARCH_MODE 型には以下がある:

- PS::SEARCH_MODE_LONG
- PS::SEARCH_MODE_LONG_CUTOFF
- PS::SEARCH_MODE_GATHER
- PS::SEARCH_MODE_SCATTER
- PS::SEARCH_MODE_SYMMETRY
- PS::SEARCH_MODE_LONG_SCATTER
- PS::SEARCH_MODE_LONG_SYMMETRY
- PS::SEARCH_MODE_LONG_CUTOFF_SCATTER
- PS::SEARCH_MODE_LONG_PARTICLE_MESH_MULTIPOLE

以下で、それぞれが対応する相互作用のモードについて記述する。

6.7.2 PS::SEARCH_MODE_LONG

この型を使用するのは、遠くの粒子からの寄与を複数の粒子にまとめた超粒子からの寄与として計算する場合である。開放境界条件における重力やクーロン力に適用できる (周期境界条件では使用不可能)。

6.7.3 PS::SEARCH_MODE_LONG_CUTOFF

この型を使用するのは、遠くの粒子からの寄与を複数の粒子にまとめた超粒子からの寄与として計算し、かつ有限の距離までの寄与しか計算しない場合である。周期境界条件における重力やクーロン力 (Particle Mesh 法の並用が必要) などに適用できる。

6.7.4 PS::SEARCH_MODE_GATHER

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が i 粒子の大きさで決まる場合である。

6.7.5 PS::SEARCH_MODE_SCATTER

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が j 粒子の大きさで決まる場合である。

6.7.6 PS::SEARCH_MODE_SYMMETRY

この型を使用するのは、相互作用の到達距離が有限でかつ、その到達距離が i, j 粒子のうち大きいほうのサイズで決まる場合である。

6.7.7 PS::SEARCH_MODE_LONG_SCATTER

基本的には SEARCH_MODE_LONG と同じであるが、 i 粒子と j 粒子の距離が j 粒子の探索半径よりも短い場合は、その j 粒子は超粒子に含めない (超粒子としてではなく粒子として扱う)。

6.7.8 PS::SEARCH_MODE_LONG_SYMMETRY

基本的には SEARCH_MODE_LONG と同じであるが、 i 粒子と j 粒子の距離が、 i 粒子と j 粒子の探索半径のどちらか大きい方よりも短い場合は、その j 粒子は超粒子に含めない (超粒子としてではなく粒子として扱う)。

6.7.9 PS::SEARCH_MODE_LONG_CUTOFF_SCATTER

未実装。

6.7.10 PS::SEARCH_MODE_LONG_PARTICLE_MESH_MULTIPOLE

PMMM を使う場合に指定するモード。このモードを指定する事で、近傍セルに所属する粒子のみが探索される。

6.8 列挙型

6.8.1 概要

本節ではFDPSで定義されている列挙型について記述する。列挙型にはBOUNDARY_CONDITION型とINTERACTION_LIST_MODE型が存在する。以下、各列挙型について記述する。

6.8.2 PS::BOUNDARY_CONDITION 型

6.8.2.1 概要

BOUNDARY_CONDITION 型は境界条件を指定するためのデータ型である。これは以下のように定義されている。

ソースコード 17: boundarycondition

```
1 namespace ParticleSimulator{
2     enum BOUNDARY_CONDITION{
3         BOUNDARY_CONDITION_OPEN ,
4         BOUNDARY_CONDITION_PERIODIC_X ,
5         BOUNDARY_CONDITION_PERIODIC_Y ,
6         BOUNDARY_CONDITION_PERIODIC_Z ,
7         BOUNDARY_CONDITION_PERIODIC_XY ,
8         BOUNDARY_CONDITION_PERIODIC_XZ ,
9         BOUNDARY_CONDITION_PERIODIC_YZ ,
10        BOUNDARY_CONDITION_PERIODIC_XYZ ,
11        BOUNDARY_CONDITION_SHEARING_BOX ,
12        BOUNDARY_CONDITION_USER_DEFINED ,
13    };
14 }
```

以下にどの変数がどの境界条件に対応するかを記述する。

6.8.2.2 PS::BOUNDARY_CONDITION_OPEN

開放境界となる。

6.8.2.3 PS::BOUNDARY_CONDITION_PERIODIC_X

x 軸方向のみ周期境界、その他の軸方向は開放境界となる。周期の境界の下限は閉境界、上限は開境界となっている。この境界の規定はすべての軸方向にあてはまる。

6.8.2.4 PS::BOUNDARY_CONDITION_PERIODIC_Y

y 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.5 PS::BOUNDARY_CONDITION_PERIODIC_Z

z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.6 PS::BOUNDARY_CONDITION_PERIODIC_XY

x, y 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.7 PS::BOUNDARY_CONDITION_PERIODIC_XZ

x, z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.8 PS::BOUNDARY_CONDITION_PERIODIC_YZ

y, z 軸方向のみ周期境界、その他の軸方向は開放境界となる。

6.8.2.9 PS::BOUNDARY_CONDITION_PERIODIC_XYZ

x, y, z 軸方向すべてが周期境界となる。

6.8.2.10 PS::BOUNDARY_CONDITION_SHEARING_BOX

未実装。

6.8.2.11 PS::BOUNDARY_CONDITION_USER_DEFINED

未実装。

6.8.3 PS::INTERACTION_LIST_MODE 型

6.8.3.1 概要

INTERACTION_LIST_MODE 型は相互作用リストを再利用するかどうかを決定するためのデータ型である。これは以下のように定義されている。

ソースコード 18: interaction_list_mode

```
1 namespace ParticleSimulator{
2     enum INTERACTION_LIST_MODE{
3         MAKE_LIST,
4         MAKE_LIST_FOR_REUSE,
5         REUSE_LIST,
6     };
7 }
```

このデータ型は calcForceAllAndWriteBack() 等の関数の引数として使われる (詳しくはセクション 9.1.4.2.3 を参照)。

6.8.3.2 PS::MAKE_LIST

相互作用リストを毎回作り相互作用計算を行う場合に用いる。FDPS は相互作用リストを記憶しないため、相互作用リストの再利用はできない。これはデフォルト動作である。

6.8.3.3 PS::MAKE_LIST_FOR_REUSE

相互作用リストを再利用し相互作用計算を行いたい場合に用いる。このオプションを選択する事で FDPS は相互作用リストを作りそれを保持する。作成した相互作用リストは PS::MAKE_LIST_FOR_REUSE もしくは PS::MAKE_LIST を用いて相互作用計算を行った際に破棄される。

6.8.3.4 PS::REUSE_INTERACTION_LIST

相互作用リストを再利用し相互作用計算を行う。再利用される相互作用リストは PS::MAKE_LIST_FOR_REUSE を選択時に作成した相互作用リストである。この時、超粒子のモーメント情報は自動的にアップデートされる。

6.8.4 PS::EXCHANGE_LET_MODE 型

6.8.4.1 概要

EXCHANGE_LET_MODE 型は LET 交換の方法を決定するためのデータ型である。これは以下のように定義されている。

ソースコード 19: EXCHANGE_LET_MODE

```
1 namespace ParticleSimulator{
2     enum EXCHANGE_LET_MODE{
3         EXCHANGE_LET_A2A ,
4         EXCHANGE_LET_P2P_EXACT ,
5         EXCHANGE_LET_P2P_FAST ,
6     };
7 }
```

このデータ型は PS::TreeForForce::setExchangeLet の引数として使われる。(詳しくはセクション 9.1.4.2.1.4 を参照)。

6.8.4.2 PS::EXCHANGE_LET_A2A

LET の交換に MPI_Alltoall を使用する。

6.8.4.3 PS::EXCHANGE_LET_P2P_EXACT

LET の交換に MPI_Alltoall を使用せず、MPI_Allgather と MPI_Isend/recv を使用する。MPI_Alltoall が効率的に動かない計算機では、こちらの方が速い場合がある。結果は丸め誤差の範囲で PS::EXCHANGE_LET_A2A を用いた場合と一致する。

6.8.4.4 EXCHANGE_LET_P2P_FAST

LET の交換に MPI_Alltoall を使用せず、MPI_Allgather と MPI_Isend/recv を使用する。結果は PS::EXCHANGE_LET_P2P_EXACT を用いた場合と異なるが、より通信量が減っているため、高速に動作する可能性がある。

6.8.5 PS::CALC_DISTANCE_TYPE 型

6.8.5.1 概要

CALC_DISTANCE_TYPE 型は粒子間の距離の計算方法を変更するための型である。

ソースコード 20: CALC_DISTANCE_TYPE

```
1 namespace ParticleSimulator{
2     enum CALC_DISTANCE_TYPE{
3         CALC_DISTANCE_TYPE_NORMAL = 0,
4         CALC_DISTANCE_TYPE_NEAREST_X = 1,
5         CALC_DISTANCE_TYPE_NEAREST_Y = 2,
6         CALC_DISTANCE_TYPE_NEAREST_XY = 3,
7         CALC_DISTANCE_TYPE_NEAREST_Z = 4,
8         CALC_DISTANCE_TYPE_NEAREST_XZ = 5,
9         CALC_DISTANCE_TYPE_NEAREST_YZ = 6,
10        CALC_DISTANCE_TYPE_NEAREST_XYZ = 7,
11    };
12 }
```

このデータ型は PS::TreeForForce のテンプレート引数として与えられる。(セクション 9.1.4.1 を参照)。

6.8.5.2 CALC_DISTANCE_TYPE_NORMAL

領域内の粒子の L2 ノルムを計算する。特に指定のない場合は、この値が使われる。

6.8.5.3 CALC_DISTANCE_TYPE_NEAREST_X

x 方向に周期境界だとした場合に最も近い粒子との L2 ノルムを計算する。FDPS の x 方向の境界条件が開放境界の場合でも、最近接粒子との距離を計算する。

6.8.5.4 CALC_DISTANCE_TYPE_NEAREST_Y

Y 方向に周期境界だとした場合に最も近い粒子との L2 ノルムを計算する。FDPS の y 方向の境界条件が開放境界の場合でも、最近接粒子との距離を計算する。

6.8.5.5 CALC_DISTANCE_TYPE_NEAREST_Z

Z 方向に周期境界だとした場合に最も近い粒子との L2 ノルムを計算する。FDPS の z 方向の境界条件が開放境界の場合でも、最近接粒子との距離を計算する。

6.8.5.6 CALC_DISTANCE_TYPE_NEAREST_XY

X,Y 方向に周期境界だとした場合に最も近い粒子との L2 ノルムを計算する。FDPS の x,y 方向の境界条件が開放境界の場合でも、最近接粒子との距離を計算する。

6.8.5.7 CALC_DISTANCE_TYPE_NEAREST_XZ

X,Z 方向に周期境界だとした場合に最も近い粒子との L2 ノルムを計算する。FDPS の x,z 方向の境界条件が開放境界の場合でも、最近接粒子との距離を計算する。

6.8.5.8 CALC_DISTANCE_TYPE_NEAREST_YZ

Y,Z 方向に周期境界だとした場合に最も近い粒子との L2 ノルムを計算する。FDPS の y,z 方向の境界条件が開放境界の場合でも、最近接粒子との距離を計算する。

6.8.5.9 CALC_DISTANCE_TYPE_NEAREST_XYZ

X,Y,Z 方向に周期境界だとした場合に最も近い粒子との L2 ノルムを計算する。FDPS の x,y,z 方向の境界条件が開放境界の場合でも、最近接粒子との距離を計算する。

6.9 PS::TimeProfile

6.9.1 概要

本節では、PS::TimeProfile 型について記述する。PS::TimeProfile 型は FDPS で使われる 3 つのクラス、領域分割クラス、粒子群クラス、相互作用ツリークラス、各メソッドの計算時間を格納するクラスである。これら 3 つのクラスには PS::TimeProfile getTimeProfile() というメソッドが存在し、このメソッドをつかってユーザーは各メソッドの計算時間を取得出来る。

このクラスは以下のように記述されている。

ソースコード 21: TimeProfile

```
1 namespace ParticleSimulator{
2 struct TimeProfile{
3     // for decompose domain
4     F64 decompose_domain_all;
5     F64 select_sample_particle;
6     F64 decompose_domain;
7     F64 decompose_domain_communication;
8     F64 decompose_domain_sort_sample;
9
10    // for exchange particle
11    F64 exchange_particle;
12    F64 exchange_particle_search;
13    F64 exchange_particle_communication;
14
15    // for interaction calculation
16    F64 calc_force_all;
17    F64 set_root_cell;
18    F64 construct_lt;
19    F64 set_particle_lt;
20    F64 sort_particle_lt;
21    F64 link_cell_lt;
22    F64 set_moment_lt;
23    F64 map_pm_cell_lt;
24    F64 construct_exchange_let;
25    F64 construct_let_1;
26    F64 let_communication_1;
27    F64 construct_let_2;
28    F64 let_communication_2;
29    F64 construct_gt;
30    F64 set_particle_gt;
31    F64 sort_particle_gt;
32    F64 link_cell_gt;
33    F64 set_moment_gt;
34    F64 map_pm_cell_gt;
35    F64 construct_ipg;
36    F64 construct_interaction_list;
37    F64 calculate_force;
38    F64 write_back;
39 };
```

メンバ変数は `public` になっているため、ユーザーは直接アクセスして値を取得することができる。このメンバ変数は Version8 のものであり、バージョンによって異なる場合があるため、各バージョンでの仕様を確認する必要がある。

6.9.1.1 メンバ変数

- `decompose_domain_all` — サンプル粒子の収集，領域の分割および通信にかかった時間
この時間はさらに以下のように分解される。
 - `select_sample_particle` — サンプル粒子を選び出すのににかかった時間
 - `decompose_domain` — サンプル粒子の収集，領域の分割にかかった時間
 - * `decompose_domain_communication` — サンプル粒子の収集や分割領域の放送などの通信にかかった時間
 - * `decompose_domain_sort_sample` — サンプル粒子のソートにかかった時間
- `exchange_particle` — 交換する粒子の収集，および通信にかかった時間
この時間はさらに以下のように分解される。
 - `exchange_particle_search` — 交換する粒子の探索にかかった時間
 - `exchange_particle_communication` — 粒子交換の通信にかかった時間
- `calc_force_all` — ローカルツリー構築，LET 交換，グローバルツリー構築，相互作用計算にかかった時間
この時間はさらに以下のように分解される。
 - `set_root_cell` — ルートセルの設定にかかった時間
 - `construct_lt` — ローカルツリーの構築，モーメント計算にかかった時間
 - * `set_particle_lt` — ローカルツリーを構成する粒子のセットにかかった時間
 - * `sort_particle_lt` — ローカルツリーを構成する EP のソートにかかった時間
 - * `link_cell_lt` — ローカルツリーセルのリンクにかかった時間
 - * `set_moment_lt` — ローカルツリーのモーメント計算にかかった時間
 - * `map_pm_cell_lt` — PM セルインデックスからローカルツリーセルインデックスのマップを計算するのににかかった時間
 - `construct_exchange_let` — LET 構築および LET 交換にかかった時間
 - * `construct_let_1` — 1 回目の LET の構築にかかった時間
 - * `let_communication_1` — 1 回目の LET の交換にかかった時間
 - * `construct_let_2` — 2 回目の LET の構築にかかった時間
 - * `let_communication_2` — 2 回目の LET の交換にかかった時間

- `construct_gt` — グローバルツリーの構築，モーメント計算にかかった時間
 - * `set_particle_gt` — グローバルツリーを構成する粒子のセットにかかった時間
 - * `sort_particle_gt` — グローバルツリーを構成する EP のソートにかかった時間
 - * `link_cell_gt` — グローバルツリーセルのリンクにかかった時間
 - * `set_moment_gt` — グローバルツリーのモーメント計算にかかった時間
 - * `map_pm_cell_gt` — PM セルインデックスからグローバルツリーセルインデックスのマップを計算するのににかかった時間
- `construct_ipg` — *i* 粒子グループの構築にかかった時間
- `construct_interaction_list` — 相互作用リストの構築にかかった時間
- `calculate_force` — 相互作用計算にかかった時間
- `write_back` — 計算した力を粒子群クラスに書き戻すのににかかった時間

6.9.1.2 加算

```
PS::TimeProfile PS::TimeProfile::operator +
    (const PS::TimeProfile & rhs) const;
```

- 引数

`rhs`: 入力 . `const TimeProfile &型` .
- 返り値

`PS::TimeProfile 型` . `rhs` のすべてのメンバ変数の値と自身のメンバ変数の値の和を取った値を返す .

6.9.1.3 除算

```
template <typename T>
TimeProfile operator/(const T n);
```

- 引数

`n`: 入力 . `const T 型` .
- 返り値

`PS::TimeProfile 型` . すべてのメンバ変数の値を `n` で割った値を返す .

6.9.1.4 乗算

```
template <typename T>
TimeProfile operator*(const T n);
```

- 引数

n: 入力 . const T 型 .

- 返回值

PS::TimeProfile 型 . すべてのメンバ変数の値を n 倍した値を返す .

6.9.1.5 縮約

```
PS::F64 PS::TimeProfile::getTotalTime() const;
```

- 引数

なし .

- 返回值

decompose_domain_all, exchange_particle, calc_force_all, write_back の合計値を返す .

6.9.1.6 初期化

```
void PS::TimeProfile::clear();
```

- 引数

なし .

- 返回值

なし .

- 機能

すべてのメンバ変数に 0 を代入する .

6.9.1.7 出力

```
void PS::TimeProfile::dump(std::ostream & fout=std::cout) const;
```

- 引数
fout: 入力, std::ostream 型, 出力先のストリームを指定する. デフォルトは std::cout .
- 戻り値
なし .
- 機能
すべてのメンバ変数の値を出力する .

7 ユーザー定義クラス・ユーザー定義関数オブジェクト

7.1 概要

本節では、ユーザーが定義するクラスとファンクタについて記述する。ユーザー定義クラスとなるのは、FullParticle クラス、EssentialParticleI クラス、EssentialParticleJ クラス、Moment クラス、SuperParticleJ クラス、Force クラス、ヘッダクラスである。またユーザー定義の関数オブジェクトには、関数オブジェクト calcForceEpEp、calcForceSpEp がある。

FullParticle クラスは、ある 1 粒子の情報すべてを持つクラスであり、粒子群クラスにテンプレート引数として渡されるものである (節 2.3 の手順 0)。

関数オブジェクト calcForceEpEp と calcForceSpEp は、それぞれ j 粒子から i 粒子への作用を計算する関数オブジェクトと超粒子から i 粒子への作用を計算する関数オブジェクトである。これらは相互作用ツリークラスの API の引数として渡されるものである (節 2.3 の手順 0)。超粒子を必要とする PS::SEARCH_MODE 型 (PS::SEARCH_MODE_LONG か PS::SEARCH_MODE_LONG_CUTOFF) 以外を使用する場合には、関数オブジェクト calcForceSpEp を定義する必要はない。

EssentialParticleI クラス、EssentialParticleJ クラス、Moment クラス、SuperParticleJ クラス、Force クラスは粒子間の相互作用の定義を補助するものである。これらのクラスのうち EssentialParticleI クラス、EssentialParticleJ クラス、Force クラスはそれぞれ相互作用を計算する際に i 粒子に必要な情報、相互作用を計算する際に j 粒子に必要な情報、相互作用の結果の情報を持つ。これらは FullParticle クラスのサブセットであるため、これらを FullParticle クラスで代用することも可能である。しかし、FullParticle クラスは相互作用の定義に必要なデータを多く含む場合も考えられるため、計算コストを軽減したいならば、これらのクラスを使用することを検討するべきである。Moment クラスと SuperParticleJ クラスは、それぞれツリーセルのモーメント情報を持つクラスと超粒子に必要な情報を持つ SuperParticleJ クラスである。ユーザーが定義する必要があるのは、超粒子を使う必要がある場合、すなわち PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG か PS::SEARCH_MODE_LONG_CUTOFF を選んだ場合のみである。

ヘッダクラスは入出力ファイルのヘッダ情報を持つ。

この節で記述するのは、これらのクラスや関数オブジェクトを定義する際の規定である。ユーザーはこれらの間でのデータのやりとりや、関数オブジェクト内でのデータの加工についてコードに書く必要がある。これらは上に挙げたクラスのメンバ関数と関数オブジェクト内で行われる。以下、必要なメンバ関数とその規定について記述する。

7.2 FullParticle クラス

7.2.1 概要

FullParticle クラスは粒子情報すべてを持つクラスであり、節 2.3 の手順 0 で、粒子群クラスに渡されるユーザー定義クラスの 1 つである。ユーザーはこのクラスに対して、どのようなメンバ変数、メンバ関数を定義してもかまわない。ただし、FDPS から FullParticle クラ

スの情報にアクセスするために、ユーザーはいくつかの決まった名前のメンバ関数を定義する必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.2.2 前提

この節の中では、以下のように、FullParticle クラスとして FP というクラスを一例とする。FP という名前は自由に変えることができる。

```
class FP;
```

7.2.3 必要なメンバ関数

7.2.3.1 概要

常に必要なメンバ関数は FP::getPos と FP::copyFromForce である。FP::getPos は FullParticle の位置情報を FDPS に読み込ませるための関数で、FP::copyFromForce は計算された相互作用の結果を FullParticle に書き戻す関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.2.3.2 FP::getPos

```
class FP {  
public:  
    PS::F64vec getPos() const;  
};
```

- 引数
なし

- 返値
PS::F32vec 型または PS::F64vec 型。FP クラスのオブジェクトの位置情報を保持したメンバ変数。

- 機能
FP クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

7.2.3.3 FP::copyFromForce

```
class Force;

class FP {
public:
    void copyFromForce(const Force & force);
};
```

- 引数

force: 入力。const Force &型。粒子の相互作用の計算結果を保持。

- 返値

なし。

- 機能

粒子の相互作用の計算結果を FP クラスへ書き戻す。

7.2.4 場合によっては必要なメンバ関数

7.2.4.1 概要

本節では、以下に示す場合に必要となるメンバ関数について記述する:

1. 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合
2. 粒子群クラスのファイル入出力 API を用いる場合
3. 粒子群クラスの API である ParticleSystem::adjustPositionIntoRootDomain を用いる場合
4. 拡張機能の Particle Mesh クラスを用いる場合

7.2.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

7.2.4.2.1 FP::getRSearch

```
class FP {
public:
    PS::F64 getRSearch() const;
};
```

- 引数

なし

- 返値

PS::F32 型または PS::F64 型。FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数。

- 機能

FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

7.2.4.3 粒子群クラスのファイル入出力 API を用いる場合

粒子群クラスのファイル入出力 API である `ParticleSystem::readParticleAscii`, `ParticleSystem::writeParticleAscii`, `ParticleSystem::readParticleBinary`, `ParticleSystem::writeParticleBinary` を使用するときそれぞれ `readAscii`, `writeAscii`, `readBinary`, `writeBinary` というメンバ関数が必要となる (`readAscii`, `writeAscii`, `readBinary`, `writeAscii` 以外の名前を使うことも可能。詳しくは節 9.1.3.2.3 を参照)。以下、`readAscii` と `writeAscii`, `readBinary`, `writeBinary` の規定について記述する。

7.2.4.3.1 *FP::readAscii*

```
class FP {  
public:  
    void readAscii(FILE *fp);  
};
```

- 引数

fp: FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

アスキー形式の粒子データの入力ファイル 1 行から FP クラス 1 個の情報を読み取り、メンバ変数に値を入れる。ここで 1 行とは、FP クラス 1 個のデータの末尾に必ず唯一つの改行コード (`\n`) がある状態の意味である。

7.2.4.3.2 *FP::writeAscii*

```
class FP {  
public:  
    void writeAscii(FILE *fp);  
};
```

- 引数

fp: FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

粒子データの出力ファイルへ FP クラス 1 個の情報を 1 行としてアスキー形式で書き出す。ここで 1 行とは、FP クラス 1 個のデータの末尾に必ず唯一つの改行コード (\n) がある状態の意味である。

7.2.4.3.3 *FP::readBinary*

```
class FP {  
public:  
    void readBinary(FILE *fp);  
};
```

- 引数

fp: FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

バイナリ形式の粒子データの入力ファイルから FP クラス 1 個の情報を読み取り、メンバ変数に値を入れる。

7.2.4.3.4 *FP::writeBinary*

```
class FP {  
public:  
    void writeBinary(FILE *fp);  
};
```

- 引数

fp: FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

粒子データの出力ファイルへ FP クラス 1 個の情報をバイナリ形式で書き出す。

7.2.4.4 *ParticleSystem::adjustPositionIntoRootDomain* を用いる場合

7.2.4.4.1 *FP::setPos*

```
class FP {  
public:  
    void setPos(const PS::F64vec pos_new);  
};
```

- 引数

pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- 返値

なし。

- 機能

FDPS が修正した粒子の位置情報を FP クラスのオブジェクトの位置情報に書き込む。

7.2.4.5 Particle Mesh クラスを用いる場合

Particle Mesh クラスを用いる場合には、メンバ関数 *FP::getChargeParticleMesh* と *FP::copyFromForceParticleMesh* を用意する必要がある。以下にそれぞれの規定を記述する。

7.2.4.5.1 *FP::getChargeParticleMesh*

```
class FP {  
public:  
    PS::F64 getChargeParticleMesh() const;  
};
```

- 引数
なし。
- 返値
PS::F32 型または PS::F64 型。1 つの粒子の質量または電荷の変数を返す。
- 機能
1 つの粒子の質量または電荷を表すメンバ変数を返す。

7.2.4.5.2 *FP::copyFromForceParticleMesh*

```
class FP {  
public:  
    void copyFromForceParticleMesh(const PS::F32vec & acc_pm);  
};
```

- 引数
acc_pm: const PS::F32vec 型または const PS::F64vec 型。1 つの粒子の Particle Mesh による力の計算結果。
- 返値
なし。
- 機能
1 つの粒子の Particle Mesh による力の計算結果をこの粒子のメンバ変数に書き込む。

7.2.4.6 粒子交換時に粒子データをシリアル化して送る場合

粒子交換時に粒子データをシリアル化して送る場合には、メンバ関数に `FP::pack` と `FP::unpack` を用意する必要がある。以下にそれぞれの規定を記述する。

7.2.4.6.1 *FP::pack*

```
class FP {  
public:  
    static PS::S32 pack(const PS::S32 n_ptcl, const FP *ptcl[], char *buf,  
                        size_t & packed_size, const size_t max_buf_size);  
};
```

- 引数

n_ptcl: 粒子交換時に送る粒子の数。

ptcl: 送る粒子へのポインタの配列。

buf: 送信バッファの先頭アドレス。

packed_size: ユーザーがバッファへ書き込むサイズ。単位はバイト。

max_buf_size: 送信バッファの書き込み可能な領域のサイズ。単位はバイト。

- 返値

PS::S32 型。packed_size が max_buf_size を超えた場合は-1 を返す。それ以外の場合は 0 を返す。

- 機能

粒子交換時に送信する粒子をシリアル化し、送信バッファに書き込む。

7.2.4.6.2 *FP::unPack*

```
class FP {  
public:  
    static PS::S32 unPack(const PS::S32 n_ptcl, FP ptcl[], const char *buf);  
};
```

- 引数

n_ptcl: 粒子交換時に受け取る粒子の数。

ptcl: 受け取る粒子の配列。

buf: 受信バッファの先頭アドレス。

- 返値

PS::S32 型。デシリアル化に失敗した場合は-1 を返す。それ以外の場合は 0 を返す。

- 機能

粒子交換時に受信する粒子をデシリアライズし、粒子配列に書き込む。詳しくはセクション 9.1.3.2.4.1 を参照。デシリアライズに失敗した場合は `PS::Abort()` が呼ばれプログラムは終了する。

7.3 EssentialParticleI クラス

7.3.1 概要

EssentialParticleI クラスは相互作用の計算に必要な i 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleI クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。そのため、EssentialParticleI クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.3.2 前提

この節の中では、EssentialParticleI クラスとして EPI というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPI, FP というクラス名は変更可能である。

EPI と FP の宣言は以下の通りである。

```
class FP;  
class EPI;
```

7.3.3 必要なメンバ関数

7.3.3.1 概要

常に必要なメンバ関数は `EPI::getPos` と `EPI::copyfromFP` である。`EPI::getPos` は EPI クラスの位置情報を FDPS に読み込ませるための関数で、`EPI::copyFromFP` は FP クラスの情報を EPI クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.3.3.2 EPI::getPos

```
class EPI {  
public:  
    PS::F64vec getPos() const;  
};
```


- 引数

なし

- 返値

PS::F64vec 型。EPI クラスの位置情報を保持したメンバ変数。

- 機能

EPI クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

7.3.3.3 EPI::copyFromFP

```
class FP;
class EPI {
public:
    void copyFromFP(const FP & fp);
};
```

- 引数

fp: 入力。const FP &型。FP クラスの情報を持つ。

- 返値

なし。

- 機能

FP クラスの持つ 1 粒子の情報の一部を EssnetialParticleI クラスに書き込む。

7.3.4 場合によっては必要なメンバ関数

7.3.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合に必要となるメンバ関数について記述する。

7.3.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合

7.3.4.2.1 EPI::getRSearch

```
class EPI {  
public:  
    PS::F64 getRSearch() const;  
};
```

- 引数
なし
- 返値
PS::F32 型または PS::F64 型。 EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。
- 機能
EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

7.4 EssentialParticleJ クラス

7.4.1 概要

EssentialParticleJ クラスは相互作用の計算に必要な j 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleJ クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。このために、EssentialParticleJ クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

7.4.2 前提

この節の中では、EssentialParticleJ クラスとして EPJ というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPJ, FP というクラス名は変更可能である。

EPJ と FP の宣言は以下の通りである。

```
class FP;  
class EPJ;
```

7.4.3 必要なメンバ関数

7.4.3.1 概要

常に必要なメンバ関数は `EPJ::getPos` と `EPJ::copyfromFP` である。`EPJ::getPos` は `EPJ` クラスの位置情報を `FDPS` に読み込ませるための関数で、`EPJ::copyFromFP` は `FP` クラスの情報を `EPJ` クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

7.4.3.2 `EPJ::getPos`

```
class EPJ {  
public:  
    PS::F64vec getPos() const;  
};
```

- 引数
なし
- 返値
`PS::F64vec` 型。`EPJ` クラスの位置情報を保持したメンバ変数。
- 機能
`EPJ` クラスの位置情報を保持したメンバ変数を返す。

7.4.3.3 `EPJ::copyFromFP`

```
class FP;  
class EPJ {  
public:  
    void copyFromFP(const FP & fp);  
};
```

- 引数
`fp`: 入力。`const FP &`型。`FP` クラスの情報を持つ。
- 返値
なし。
- 機能
`FP` クラスの持つ 1 粒子の情報の一部を `EPJ` クラスに書き込む。

7.4.4 場合によっては必要なメンバ関数

7.4.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合に必要なメンバ関数、列挙型の BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を選んだ場合に必要となるメンバ関数について記述する。なお、既存の Moment クラスや SuperParticleJ クラスを用いる際に必要となるメンバ変数はこれら既存のクラスの節を参照のこと。

7.4.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

7.4.4.2.1 EPJ::getRSearch

```
class EPJ {  
public:  
    PS::F64 getRSearch() const;  
};
```

- 引数

なし

- 返値

PS::F32 型または PS::F64 型。 EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- 機能

EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

なお、PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG_CUTOFF を用いる場合、すべての EPJ が同じ値を返すように定義しなければならない。

7.4.4.3 BOUNDARY_CONDITION型にPS::BOUNDARY_CONDITION_OPEN 以外を用いる場合

7.4.4.3.1 EPJ::setPos

```
class EPJ {  
public:  
    void setPos(const PS::F64vec pos_new);  
};
```

- 引数

pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- 返値

なし。

- 機能

FDPS が修正した粒子の位置情報を EPJ クラスの位置情報に書き込む。

7.4.4.4 粒子の id 番号から対応する EPJ を取得したい場合

7.4.4.4.1 EPJ::getId

```
class EPJ {  
public:  
    PS::S64 getId();  
};
```

- 引数

なし。

- 返値

PS::S64 型。

- 機能

PS::TreeForForce::getEpjFromId() を使用する場合に必要。詳しくは 9.1.4.2.7 を参照。

7.4.4.5 LET 交換時に粒子データをシリアルライズして送る場合

LET 交換時に粒子データをシリアルライズして送る場合には、メンバ関数に EPJ::pack と EPJ::unpack を用意する必要がある。以下にそれぞれの規定を記述する。

7.4.4.5.1 EPJ::pack

```
class EPJ {
public:
    static PS::S32 pack(const PS::S32 n_ptcl, const EPJ *ptcl[], char *buf,
                       size_t & packed_size, const size_t max_buf_size);
};
```

- 引数

n_ptcl: LET 交換時に送る粒子の数。

ptcl: 送る粒子へのポインタの配列。

buf: 送信バッファの先頭アドレス。

packed_size: ユーザーがバッファへ書き込むサイズ。単位はバイト。

max_buf_size: 送信バッファの書き込み可能な領域のサイズ。単位はバイト。

- 返値

PS::S32 型。packed_size が max_buf_size を超えた場合は-1 を返す。それ以外の場合は 0 を返す。

- 機能

LET 交換時に送信する粒子をシリアルライズし、送信バッファに書き込む。

7.4.4.5.2 EPJ::unPack

```
class EPJ {
public:
    static void unPack(const PS::S32 n_ptcl, EPJ ptcl[], const char *buf);
};
```

- 引数

n_ptcl: LET 交換時に受け取る粒子の数。

ptcl: 受け取る粒子の配列。

buf: 受信バッファの先頭アドレス。

- 返値

なし。

- 機能

LET 交換時に受信する粒子をデシリアライズし、粒子配列に書き込む。詳しくはセクション 9.1.4.2.3 を参照。デシリアライズに失敗した場合は `PS::Abort()` が呼ばれプログラムは終了する。

7.5 Moment クラス

7.5.1 概要

Moment クラスは近い粒子同士でまとまった複数の粒子のモーメント情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。モーメント情報の例としては、複数粒子の単極子や双極子、さらにこれら粒子の持つ最大の大きさなど様々なものが考えられる。このクラスは、`EssentialParticleJ` クラスから `SuperParticleJ` クラスを作るための中間変数のような役割を果たす。従って、このクラスが持つメンバ関数は、`EssentialParticleJ` クラスから情報を読み出してモーメントを計算するメンバ関数、少ない数の粒子のモーメントからそれらの粒子を含むより多くの粒子のモーメントを計算するメンバ関数などがある。

このようなモーメント情報にはある程度決まっているものが多いので、それらについては FDPS 側で用意した。これら既存のクラスについてまず記述する。その後にユーザーがモーメントクラスを自作する際に必ず必要なメンバ関数、場合によっては必要になるメンバ関数について記述する。

7.5.2 既存のクラス

7.5.2.1 概要

FDPS はいくつかの Moment クラスを用意している。これらは相互作用ツリークラスで特定の `PS::SEARCH_MODE` 型を選んだ場合に有効である。以下、各 `PS::SEARCH_MODE` 型において選ぶことのできる Moment 型を記述する。`PS::SEARCH_MODE_GATHER`, `PS::SEARCH_MODE_SCATTER`, `PS::SEARCH_MODE_SYMMETRY` については Moment クラスを意識してコーディングする必要がないので、これらについては記述しない。

7.5.2.2 `PS::SEARCH_MODE_LONG`

7.5.2.2.1 *PS::MomentMonopole*

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopole {
    public:
        F64    mass;
        F64vec pos;
    };
}
```

- クラス名 PS::MomentMonopole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.2 PS::MomentQuadrupole

単極子と四重極子を情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupole {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
    };
}
```

- クラス名 PS::MomentQuadrupole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量

pos: 近傍でまとめた粒子の重心

quad: 近傍でまとめた粒子の四重極子

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.3 PS::MomentMonopoleGeometricCenter

単極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class MomentMonopoleGeometricCenter {  
    public:  
        F64      charge;  
        F64vec pos;  
    };  
}
```

- クラス名 PS::MomentMonopoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.4 PS::MomentDipoleGeometricCenter

双極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentDipoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
        F64vec dipole;
    };
}
```

- クラス名 PS::MomentDipoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.2.5 PS::MomentQuadrupoleGeometricCenter

四重極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
        F64vec dipole;
        F64mat quadrupole;
    };
}
```

- クラス名 PS::MomentQuadrupoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

quadrupole: 粒子の質量または電荷の四重極子

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.3 PS::SEARCH_MODE_LONG_SCATTER

7.5.2.3.1 PS::MomentMonopoleScatter

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class MomentMonopoleScatter {  
    public:  
        F64      mass;  
        F64vec pos;  
        F64ort vertex_out_;  
        F64ort vertex_in_;  
    };  
}
```

- クラス名 PS::MomentMonopoleScatter

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.3.2 PS::MomentQuadrupoleScatter

単極子と四重極子を情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentQuadrupoleScatter {
    public:
        F64      mass;
        F64vec    pos;
        F64mat    quad;
        F64ort    vertex_out_;
        F64ort    vertex_in_;
    };
}
```

- クラス名 PS::MomentQuadrupoleScatter

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

quad: 近傍でまとめた粒子の四重極子

vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.4 PS::SEARCH_MODE_LONG_SYMMETRY

7.5.2.4.1 PS::MomentMonopoleSymmetry

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```

namespace ParticleSimulator {
    class MomentMonopoleSymmetry {
    public:
        F64      mass;
        F64vec pos;
        F64ort vertex_out_;
        F64ort vertex_in_;
    };
}

```

- クラス名 PS::MomentMonopoleSymmetry

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.4.2 PS::MomentQuadrupoleSymmetry

単極子と四重極子を情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```

namespace ParticleSimulator {
    class MomentQuadrupoleSymmetry {
    public:
        F64      mass;
        F64vec pos;
        F64mat quad;
        F64ort vertex_out_;
        F64ort vertex_in_;
    };
}

```

- クラス名 PS::MomentQuadrupoleSymmetry

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

quad: 近傍でまとめた粒子の四重極子

vertex_out_: 粒子を EssentialParticleJ::getRSearch の返り値を半径とする球とみなしたとき、近傍でまとめた粒子すべてを包含する最小の直方体の座標情報

vertex_in_: 近傍でまとめた粒子すべての座標を包含する最小の直方体の座標情報

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.2.5 PS::SEARCH_MODE_LONG_CUTOFF

7.5.2.5.1 PS::MomentMonopoleCutoff

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class MomentMonopoleCutoff {
    public:
        F64      mass;
        F64vec pos;
    };
}
```

- クラス名 PS::MomentMonopoleCutoff

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

7.5.3 必要なメンバ関数

7.5.3.1 概要

以下では Moment クラスを定義する際に、必要なメンバ関数を記述する。このとき Moment クラスのクラス名を Mom とする。これは変更自由である。

7.5.3.2 コンストラクタ

```
class Mom {
public:
    Mom ();
};
```

- 引数

なし

- 返値

なし

- 機能

Mom クラスのオブジェクトの初期化をする。

7.5.3.3 Mom::init

```
class Mom {  
public:  
    void init();  
};
```

- 引数

なし

- 返値

なし

- 機能

Mom クラスのオブジェクトの初期化をする。

7.5.3.4 Mom::getPos

```
class Mom {  
public:  
    PS::F32vec getPos() const;  
};
```

- 引数

なし

- 返値

PS::F32vec または PS::F64vec 型。Mom クラスのメンバ変数 pos。

- 機能

Mom クラスのメンバ変数 pos を返す。

7.5.3.5 Mom::getCharge

```
class Mom {  
public:  
    PS::F32 getCharge() const;  
};
```

- 引数
なし
- 返値
PS::F32 または PS::F64 型。Mom クラスのメンバ変数 `mass`。
- 機能
Mom クラスのメンバ変数 `mass` を返す。

7.5.3.6 Mom::getVertexIn

```
class Mom {  
public:  
    PS::F32ort getVertexIn() const;  
};
```

- 引数
なし
- 返値
PS::F32ort または PS::F64ort 型。
- 機能
Mom クラスに対応する (複数の) 粒子すべてを包含する直方体の

7.5.3.7 Mom::accumulateAtLeaf

```
class Mom {  
public:  
    template <class Tepj>  
    void accumulateAtLeaf(const Tepj & epj);  
};
```

- 引数

epj: 入力。const Tepj &型。Tepj のオブジェクト。

- 返値

なし。

- 機能

EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

7.5.3.8 Mom::accumulate

```
class Mom {  
public:  
    void accumulate(const Mom & mom);  
};
```

- 引数

mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- 返値

なし。

- 機能

Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

7.5.3.9 Mom::set

```
class Mom {  
public:  
    void set();  
};
```

- 引数

なし

- 返値

なし

- 機能

上記のメンバ関数 `Mom::accumulateAtLeaf`, `Mom::accumulate` ではモーメントの位置情報の規格化ができていない場合なので、ここで規格化する。

7.5.3.10 `Mom::accumulateAtLeaf2`

```
class Mom {
public:
    template <class Tepj>
    void accumulateAtLeaf2(const Tepj & epj);
};
```

- 引数

epj: 入力。const Tepj &型。Tepj のオブジェクト。

- 返値

なし。

- 機能

EssentialParticleJ クラスのオブジェクトからモーメントを計算する。

7.5.3.11 `Mom::accumulate2`

```
class Mom {
public:
    void accumulate2(const Mom & mom);
};
```

- 引数

mom: 入力。const Mom &型。Mom クラスのオブジェクト。

- 返値

なし。

- 機能

Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

7.5.4 場合によっては必要なメンバ関数

7.5.4.1 概要

以下では Moment クラスを定義する際に、場合によっては必要なメンバ関数を記述する。このとき Moment クラスのクラス名を Mom とする。これは変更自由である。

7.5.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG_CUTOFF、 PS::SEARCH_MODE_LONG_SCATTER、 PS::SEARCH_MODE_LONG_SYMMETRY のいずれかを用いる場合

7.5.4.3 Mom::getVertexIn

```
class Mom {  
public:  
    F64ort getVertexIn();  
};
```

- 引数

なし。

- 返値

PS::F32ort または PS::F64ort 型。

- 機能

Mom クラスに対応する粒子すべてを包含する最小の直方体 (2 次元の場合には直方体) の座標をオルソトープ型で返す。

- 備考

上記の直方体の座標は、各ツリーセルで正しく計算される必要がある。そのためには、粒子情報からツリー構造の一番深いレベルのツリーセルのモーメント情報を計算するためのメンバ関数 `accumulateAtLeaf` と、複数のツリーセルからその親セルのモーメント情報を計算するためのメンバ関数 `accumulate` に直方体の座標計算を実装しなければならない。以下に実装例を示す。

```

class Mom {
public:
    F64ort vertex_in_; // 直方体の座標を格納するメンバ変数
    F64ort getVertexIn() const { return vertex_in_; }
    template<class Tepj>
    void accumulateAtLeaf(const Tepj & epj){
        // 他の演算は省略
        (this->vertex_in_).merge(epj.getPos());
    }
    void accumulate(const Mom & mom){
        // 他の演算は省略
        (this->vertex_in_).merge(mom.vertex_in_);
    }
};

```

上記実装例では、オルソトープ型のメンバ関数 `merge` を使用している。なお、メンバ変数 `vertex_in_` の名称は自由に変更可能である。

7.5.4.4 Mom::getVertexOut

```

class Mom {
public:
    F64ort getVertexOut();
};

```

- 引数
なし。

- 返値
PS::F32ort または PS::F64ort 型。

- 機能
Mom クラスに対応する各粒子を、中心を粒子座標、半径をその粒子の `getRSearch()` の返り値とする球 (2 次元の場合には円) に置き換えたとき、それらすべてを包含する最小の直方体 (2 次元の場合には直方体) の座標をオルソトープ型で返す。

- 備考
メンバ関数 `getVertexIn` のときと同様、上記の直方体の座標は、各ツリーセルで正しく計算される必要がある。そのためには、メンバ関数 `accumulateAtLeaf` とメンバ関数 `accumulate` に直方体の座標計算を実装しなければならない。以下に実装例を示す。

```

class Mom {
public:
    F64ort vertex_out_; // 直方体の座標を格納するメンバ変数
    F64ort getVertexOut() const { return vertex_out_; }
    template<class Tepj>
    void accumulateAtLeaf(const Tepj & epj){
        // 他の演算は省略
        (this->vertex_out_).merge(epj.getPos(), epj.getRSearch());
    }
    void accumulate(const Mom & mom){
        // 他の演算は省略
        (this->vertex_out_).merge(mom.vertex_out_);
    }
};

```

上記実装例では、オルソトープ型のメンバ関数 `merge` を使用している。なお、メンバ変数 `vertex_out_` の名称は自由に変更可能である。

7.6 SuperParticleJ クラス

7.6.1 概要

SuperParticleJ クラスは近い粒子同士でまとめた複数の粒子を代表してまとめた超粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。このクラスが必要となるのは `PS::SEARCH_MODE` に以下のいずれかを選択した場合だけである。

- `PS::SEARCH_MODE_LONG`
- `PS::SEARCH_MODE_LONG_SCATTER`
- `PS::SEARCH_MODE_LONG_SYMMETRY`
- `PS::SEARCH_MODE_LONG_CUTOFF`

このクラスのメンバ関数には、超粒子の位置情報を FDPS 側とやりとりするメンバ関数がある。また、超粒子の情報と Moment クラスの情報は対になるものである。従って、このクラスのメンバ関数には、Moment クラスからこのクラスへ情報を変換 (またはその逆変換) するメンバ関数がある。

SuperParticleJ クラスも Moment クラス同様、ある程度決っているものが多いので、それらについては FDPS 側で用意した。以下、既存のクラス、SuperParticleJ クラスを作るときに必要なメンバ関数、場合によっては必要なメンバ関数について記述する。

7.6.2 既存のクラス

FDPS はいくつかの SuperParticleJ クラスを用意している。以下、各 PS::SEARCH_MODE に対し選ぶことのできるクラスについて記述する。PS::SEARCH_MODE_LONG、PS::SEARCH_MODE_LONG_SYMMETRY、PS::SEARCH_MODE_LONG_CUTOFF の場合をこの順序で記述する。その他の PS::SEARCH_MODE では超粒子を必要としない。

7.6.2.1 PS::SEARCH_MODE_LONG

7.6.2.1.1 PS::SPJMonopole

単極子までの情報を持つ Moment クラス PS::MomentMonopole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class SPJMonopole {  
    public:  
        F64      mass;  
        F64vec   pos;  
    };  
}
```

- クラス名 PS::SPJMonopole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

Moment クラスである PS::MomentMonopole クラスの使用条件に準ずる。

7.6.2.1.2 PS::SPJQuadrupole

単極子と四重極子を情報を持つ Moment クラス PS::MomentQuadrupole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupole {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
    };
}
```

- クラス名 PS::SPJQuadrupole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

quad: 近傍でまとめた粒子の四重極子

- 使用条件

Moment クラスである PS::MomentQuadrupole クラスの使用条件に準ずる。

7.6.2.1.3 PS::SPJMonopoleGeometricCenter

単極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心) Moment クラス PS::MomentMonopoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleGeometricCenter {
    public:
        F64    charge;
        F64vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

- 使用条件

PS::MomentMonopoleGeometricCenter の使用条件に準ずる。

7.6.2.1.4 PS::SPJDipoleGeometricCenter

双極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心) Moment クラス PS::MomentDipoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJDipoleGeometricCenter {
    public:
        F64      charge;
        F64vec pos;
        F64vec dipole;
    };
}
```

- クラス名 PS::SPJDipoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

- 使用条件

PS::MomentDipoleGeometricCenter の使用条件に準ずる。

7.6.2.1.5 PS::SPJQuadrupoleGeometricCenter

四重極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心) Moment クラス PS::MomentQuadrupoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupoleGeometricCenter {
    public:
        F64      charge;
        F64vec pos;
        F64vec dipole;
        F64mat quadrupole;
    };
}
```

- クラス名 PS::SPJQuadrupoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

quadrupole: 粒子の質量または電荷の四重極子

- 使用条件

PS::MomentQuadrupoleGeometricCenter の使用条件に準ずる。

7.6.2.2 PS::SEARCH_MODE_LONG_SCATTER

7.6.2.2.1 PS::SPJMonopoleScatter

単極子までを情報として持つ Moment クラス PS::MomentMonopoleScatter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleScatter {
    public:
        F64      mass;
        F64vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleScatter

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

PS::MomentMonopoleScatter の使用条件に準ずる。

7.6.2.2.2 PS::SPJQuadrupoleScatter

単極子と四重極子を情報として持つ Moment クラス PS::MomentQuadrupoleScatter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupoleScatter {
    public:
        F64    mass;
        F64vec pos;
        F64mat quad;
    };
}
```

- クラス名 PS::SPJQuadrupoleScatter

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

quad: 近傍でまとめた粒子の四重極子

- 使用条件

PS::MomentQuadrupoleScatter の使用条件に準ずる。

7.6.2.3 PS::SEARCH_MODE_LONG_SYMMETRY

7.6.2.3.1 PS::SPJMonopoleSymmetry

単極子までを情報として持つ Moment クラス PS::MomentMonopoleSymmetry と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleSymmetry {
    public:
        F64    mass;
        F64vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleSymmetry

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

PS::MomentMonopoleSymmetry の使用条件に準ずる。

7.6.2.3.2 PS::SPJQuadrupoleSymmetry

単極子と四重極子を情報として持つ Moment クラス PS::MomentQuadrupoleSymmetry と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class SPJQuadrupoleSymmetry {  
    public:  
        F64    mass;  
        F64vec pos;  
        F64mat quad;  
    };  
}
```

- クラス名 PS::SPJQuadrupoleSymmetry

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

quad: 近傍でまとめた粒子の四重極子

- 使用条件

PS::MomentQuadrupoleSymmetry の使用条件に準ずる。

7.6.2.4 PS::SEARCH_MODE_LONG_CUTOFF

7.6.2.4.1 PS::SPJMonopoleCutoff

単極子までを情報として持つ Moment クラス PS::MomentMonopoleCutoff と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class SPJMonopoleCutoff {  
    public:  
        F64    mass;  
        F64vec pos;  
    };  
}
```

- クラス名 PS::SPJMonopoleCutoff
- メンバ変数とその情報
 - mass: 近傍でまとめた粒子の全質量、または全電荷
 - pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件
 - PS::MomentMonopoleCutoff の使用条件に準ずる。

7.6.3 必要なメンバ関数

7.6.3.1 概要

以下では SuperParticleJ クラスを作る際に必要なメンバ関数を記述する。このとき SuperParticleJ クラスのクラス名を SPJ とする。これは変更自由である。

7.6.3.2 SPJ::getPos

```
class SPJ {
public:
    PS::F64vec getPos() const;
};
```

- 引数
 - なし
- 返値
 - PS::F32vec 型または PS::F64vec 型。SPJ クラスの位置情報を保持したメンバ変数。
- 機能
 - SPJ クラスの位置情報を保持したメンバ変数を返す。

7.6.3.3 SPJ::setPos

```
class SPJ {
public:
    void setPos(const PS::F64vec pos_new);
};
```

- 引数

pos_new: 入力。const PS::F32vec または const PS::F64vec 型。FDPS 側で修正した粒子の位置情報。

- 返値

なし。

- 機能

FDPS が修正した粒子の位置情報を SPJ クラスの位置情報に書き込む。

7.6.3.4 SPJ::copyFromMoment

```
class Mom;
class SPJ {
public:
    void copyFromMoment(const Mom & mom);
};
```

- 引数

mom: 入力。const Mom &型。Mom にはユーザー定義または FDPS 側で用意した Moment クラスが入る。

- 返値

なし。

- 機能

Mom クラスの情報を SPJ クラスにコピーする。

7.6.3.5 SPJ::convertToMoment

```
class Mom;
class SPJ {
public:
    Mom convertToMoment() const;
};
```

- 引数

なし

- 返値

Mom 型。Mom クラスのコンストラクタ。

- 機能

超粒子をモーメントに変換し、その変換したものを Mom クラスのコンストラクタを返す。

7.6.3.6 SPJ::clear

```
class SPJ {  
public:  
    void clear();  
};
```

- 引数

なし

- 返値

なし

- 機能

SPJ クラスのオブジェクトの情報をクリアする。

7.6.4 場合によっては必要なメンバ関数

7.6.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。

7.6.4.2 LET 交換時に粒子データをシリアルライズして送る場合

LET 交換時に粒子データをシリアルライズして送る場合には、メンバ関数に SPJ::pack と SPJ::unpack を用意する必要がある。以下にそれぞれの規定を記述する。

7.6.4.2.1 SPJ::pack

```
class SPJ {  
public:  
    static PS::S32 pack(const PS::S32 n_ptcl, const SPJ *ptcl[], char *buf,  
                        size_t & packed_size, const size_t max_buf_size);  
};
```

- 引数

n_ptcl: LET 交換時に送る粒子の数。

ptcl: 送る粒子へのポインタの配列。

buf: 送信バッファの先頭アドレス。

packed_size: ユーザーがバッファへ書き込むサイズ。単位はバイト。

max_buf_size: 送信バッファの書き込み可能な領域のサイズ。単位はバイト。

- 返値

PS::S32 型。packed_size が max_buf_size を超えた場合は-1 を返す。それ以外の場合は 0 を返す。

- 機能

LET 交換時に送信する粒子をシリアルライズし、送信バッファに書き込む。

7.6.4.2.2 SPJ::unPack

```
class SPJ {  
public:  
    static void unPack(const PS::S32 n_ptcl, SPJ[], const char *buf);  
};
```

- 引数

n_ptcl: LET 交換時に受け取る粒子の数。

ptcl: 受け取る粒子の配列。

buf: 受信バッファの先頭アドレス。

- 返値

なし。

- 機能

LET 交換時に受信する粒子をデシリアルライズし、粒子配列に書き込む。

7.7 Force クラス

7.7.1 概要

Force クラスは相互作用の結果を保持するクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、この節の前提、常に必要なメンバ関数について記述する。

7.7.2 前提

この節で用いる例として Force クラスのクラス名を Result とする。このクラス名は変更自由である。

7.7.3 必要なメンバ関数

常に必要なメンバ関数は Result::clear である。この関数は相互作用の計算結果を初期化する。以下、Result::clear について記述する。

7.7.3.1 Result::clear

```
class Result {  
public:  
    void clear();  
};
```

- 引数
なし
- 返値
なし。
- 機能
Result クラスのメンバ変数を初期化する。

7.8 ヘッダクラス

7.8.1 概要

ヘッダクラスは入出力ファイルのヘッダの形式を決めるクラスである。ヘッダクラスはFDPS が提供する粒子群クラスのファイル入出力 API を使用し、かつ入出力ファイルにヘッダを含ませたい場合に必要となるクラスである。粒子群クラスのファイル入出力 API とは、ParticleSystem::readParticleAscii, ParticleSystem::writeParticleAscii, ParticleSystem::readParticleBinary,

ParticleSystem::writeParticleBinary である。以下、この節における前提と、これらの API を使用する際に必要となるメンバ関数とその記述の規定を述べる。この節において、常に必要なメンバ関数というものは存在しない。

7.8.2 前提

この節では、ヘッダクラスのクラス名を Hdr とする。このクラス名は変更可能である。

7.8.3 場合によっては必要なメンバ関数

7.8.3.1 Hdr::readAscii

```
class Hdr {  
public:  
    PS::S32 readAscii(FILE *fp);  
};
```

- 引数
fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。
- 返値
PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。
- 機能
粒子データの入力ファイルからヘッダ情報を読みこむ。

7.8.3.2 Hdr::writeAscii

```
class Hdr {  
public:  
    void writeAscii(FILE *fp);  
};
```

- 引数
fp: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。
- 返値
なし。
- 機能
粒子データの出力ファイルへヘッダ情報を書き込む。

7.8.3.3 Hdr::readBinary

```
class Hdr {  
public:  
    PS::S32 readBinary(FILE *fp);  
};
```

- 引数
fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。
- 返値
PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。
- 機能
粒子データの入力ファイルからヘッダ情報を読みこむ。

7.8.3.4 Hdr::writeBinary

```
class Hdr {  
public:  
    void writeBinary(FILE *fp);  
};
```

- 引数
fp: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。
- 返値
なし。
- 機能
粒子データの出力ファイルへヘッダ情報を書き込む。

7.9 関数オブジェクト calcForceEpEp

7.9.1 概要

関数オブジェクト calcForceEpEp は粒子同士の相互作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

7.9.2 前提

ここで示すのは重力 N 体シミュレーションの粒子間相互作用の記述の仕方である。関数オブジェクト calcForceEpEp の名前は gravityEpEp とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, EssentialParitlceJ クラスのクラス名を EPJ, Force クラスのクラス名を Result とする。

7.9.3 gravityEpEp::operator ()

ソースコード 22: calcForceEpEp

```
1 class Result;
2 class EPI;
3 class EPJ;
4 struct gravityEpEp {
5     void operator () (const EPI *epi,
6                       const PS::S32 ni,
7                       const EPJ *epj,
8                       const PS::S32 nj,
9                       Result *result);
10 };
```

- 引数

epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

epj: 入力。const EPJ *型または EPJ *型。j 粒子情報を持つ配列。

nj: 入力。const PS::S32 型または PS::S32 型。j 粒子数。

result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- 返値

なし。

- 機能

j 粒子から i 粒子への作用を計算する。

7.10 関数オブジェクト calcForceSpEp

7.10.1 概要

関数オブジェクト calcForceSpEp は超粒子から粒子への作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、この書き方の規定を記述する。

7.10.2 前提

ここで示すのは重力N体シミュレーションにおける超粒子から粒子への作用の記述の仕方である。超粒子は単極子までの情報で作られているものとする。関数オブジェクト calcForceSpEp の名前は gravitySpEp とする。これは変更自由である。また、EssentialParticleI クラスのクラス名を EPI, SuperParticleJ クラスのクラス名を SPJ, Force クラスのクラス名を Result とする。

7.10.3 gravitySpEp::operator ()

ソースコード 23: calcForceSpEp

```
1 class Result;
2 class EPI;
3 class SPJ;
4 struct gravitySpEp {
5     void operator () (const EPI *epi,
6                       const PS::S32 ni,
7                       const SPJ *spj,
8                       const PS::S32 nj,
9                       Result *result);
10 };
```

- 引数

epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

spj: 入力。const SPJ *型または SPJ *型。超粒子情報を持つ配列。

nj: 入力。const PS::S32 型または PS::S32 型。超粒子数。

result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- 返値

なし。

- 機能

超粒子から i 粒子への作用を計算する。

7.11 関数オブジェクト calcForceDispatch

7.11.1 概要

関数 calcForceDispatch は関数 calcForceRetrieve と合わせて粒子同士の相互作用を記述するものであり、calcForceSpEp や calcForceEpEp の代わりに相互作用の定義 (節 2.3 の手順 0) に使うことができる。calcForceSpEp や calcForceEpEp との違いは、calcForceDispatch は複数の相互作用リストと i 粒子リストを受け取ることである。これにより、GPGPU 等のアクセラレータを起動する回数を削減し、実行効率を向上させる。以下、この書き方の規定を記述する。関数 calcForceDispatch の名前は GravityDispatch とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, EssentialParitlceJ クラスのクラス名を EPJ, SuperParitlceJ クラスのクラス名を SPJ とする。

7.11.2 短距離力の場合

ソースコード 24: calcForceDispatch

```
1 class EPI;
2 class EPJ;
3 PS::S32 HydroforceDispatch(const PS::S32 tag,
4                             const PS::S32 nwalk,
5                             const EPI** epi,
6                             const PS::S32* ni,
7                             const EPJ** epj,
8                             const PS::S32* nj_ep;
9 };
```

- 引数

tag: 入力。const PS::S32 型。tag の番号。発行される tag の番号は 0 から関数 PS::TreeForForce::calcForceAllandWriteBackMultiWalk() の第三引数として設定された値から 1 引いた数までである。

tag の番号は calcForceRetrieve() で設定する tag の番号と対応させる必要がある。

nwalk: 入力。const PS::S32 型。walk の数。walk の数の最大値は PS::TreeForForce::calcForceAllandWriteBackMultiWalk() の第六引数の値である。

epi: 入力。const EPI** 型。i 粒子情報を持つポインタのポインタ。

ni: 入力。const PS::S32* 型。i 粒子数のポインタ。

epj: 入力。const EPJ** 型。j 粒子情報を持つポインタのポインタ。

nj_ep: 入力。const PS::S32* 型。j 粒子数のポインタ。

- 返値

PS::S32 型。ユーザーは正常に実行された場合は 0 を、エラーが起こった場合は 0 以外の値を返すようにする。

- 機能

epi,epj の情報をアクセラレータに送り、相互作用カーネルを発行する。

7.11.3 長距離力の場合

ソースコード 25: calcForceDispatch

```
1 class EPI;
2 class EPJ;
3 class SPJ;
4 PS::S32 GravityDispatch(const PS::S32 tag,
5                          const PS::S32 nwalk,
6                          const EPI** epi,
7                          const PS::S32* ni,
8                          const EPJ** epj,
9                          const PS::S32* nj_ep,
10                         const SPJ** spj,
11                         const PS::S32* nj_sp);
12 };
```

- 引数

tag: 入力。const PS::S32 型。tag の番号。発行される tag の番号は 0 から関数 PS::TreeForForce::calcForce の第三引数として設定された値から 1 引いた数までである。tag の番号は calcForceRetrieve() で設定する tag の番号と対応させる必要がある。

nwalk: 入力。const PS::S32 型。walk の数。walk の数の最大値は PS::TreeForForce::calcForceAllandWalk の第六引数の値である。

epi: 入力。const EPI** 型。i 粒子情報を持つ配列の配列。

ni: 入力。const PS::S32* 型。i 粒子数の配列。

epj: 入力。const EPJ** 型。j 粒子情報を持つ配列の配列。

nj_ep: 入力。const PS::S32* 型。j 粒子数の配列。

spj: 入力。const SPJ** 型。j 粒子情報を持つ配列の配列。

nj_sp: 入力。const PS::S32* 型。j 粒子数の配列。

- 返値

PS::S32 型。ユーザーは正常に実行された場合は 0 を、エラーが起こった場合は 0 以外の値を返すようにする。

- 機能

epi,epj,spj の情報をアクセラレータに送り、相互作用カーネルを発行する。

7.12 関数オブジェクト calcForceRetrieve

7.12.1 概要

関数 calcForceRetrieve は関数 calcForceDispatch で行った相互作用の結果を回収する関数である。以下、この書き方の規定を記述する。関数 calcForceRetrieve の名前は GravityRetrieve とする。これは変更自由である。また、Force クラスのクラス名を Result とする。

ソースコード 26: calcForceDispatch

```
1 class EPI;
2 class EPJ;
3 class Result;
4 PS::S32 GravityRetrieve(const PS::S32 tag,
5                        const PS::S32 nwalk,
6                        const PS::S32 ni [],
7                        Result result [][]);
8 };
```

- 引数

tag: 入力。const PS::S32 型。tag の番号。対応する calcForceDispatch の tag 番号と一致させる必要がある。

nwalk: 入力。const PS::S32 型。walk の数。対応する calcForceDispatch に与えた nwalk の値と一致させる必要がある。

ni: 入力。const PS::S32*型。i 粒子数の配列。

result: 出力。Result *型。i 粒子の相互作用結果を返す配列の配列。

- 返値

PS::S32 型。ユーザーは正常に実行された場合は 0 を、エラーが起こった場合は 0 以外の値を返すようにする。

- 機能

同じ tag 番号を持つ関数 calcForceDispatch で行った相互作用の結果を回収する。

8 プログラムの開始と終了

8.1 概要

プログラムの開始と終了に必要な API などを記述する。

8.2 API

8.2.1 PS::Initialize

プログラムの開始を行うには以下の API を呼び出す必要がある。

```
void PS::Initialize
    (PS::S32 & argc,
     char ** & argv,
     const PS::S64 mpool_size=100000000);
```

- 引数

argc: 入力。PS::S32 型。コマンドライン引数の総数。

argv: 入力。char ** &型。コマンドライン引数の文字列を指すポインタのポインタ。

mpool_size: 入力。const PS::S64 型。FDPS が内部で使用するメモリプールのサイズ。単位はバイト。デフォルト値は 10^8 バイト。

- 返値

なし

- 機能

FDPS の初期化を行う。FDPS の API のうち最初に呼び出さなければならない。内部では MPI::Init を呼び出すため、引数 argc と argv が変っている可能性がある。

8.2.2 PS::Finalize

プログラムの終了するには以下の API を呼び出す必要がある。

```
void PS::Finalize();
```

- 引数

なし

- 返値

なし

- 機能

FDPS の終了処理を行う。

8.2.3 PS::Abort

プログラムを異常終了させる場合には以下の API を呼び出す必要がある。

```
void PS::Abort(const PS::S32 err = -1);
```

- 引数

err: PS::S32 型。プログラムの終了ステータス。デフォルト-1。

- 返値

なし

- 機能

FDPS の異常終了処理を行う。引数はプログラムの終了ステータスである。この引数は、MPI を使用していない場合は `exit` 関数に渡され、MPI を使用している場合は MPI の `Abort` 関数に渡される。

8.2.4 PS::DisplayInfo

```
void PS::DisplayInfo();
```

- 引数

なし

- 返値

なし

- 機能

FDPS のライセンス情報などを表示する。

9 モジュール

本節では、FDPS のモジュールについて記述する。最初に FDPS の標準機能について、次に FDPS の拡張機能について記述する。

9.1 標準機能

9.1.1 概要

本節では、FDPS の標準機能について記述する。標準機能には 4 つのモジュールがあり、領域クラス、粒子群クラス、相互作用ツリークラス、通信用データクラスがある。この 4 つのクラスについて順に記述する。

9.1.2 領域クラス

本節では、領域クラスについて記述する。このクラスは領域情報の保持や領域の分割を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.2.1 オブジェクトの生成

領域クラスは以下のように宣言されている。

ソースコード 27: DomainInfo0

```
1 namespace ParticleSimulator {  
2     class DomainInfo;  
3 }
```

領域クラスのオブジェクトの生成は以下のように行う。ここでは dinfo というオブジェクトを生成している。

```
PS::DomainInfo dinfo;
```

9.1.2.2 API

領域クラスには初期設定関連の API、領域分割関連の API がある。以下、各節に分けて記述する。

9.1.2.2.1 初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 28: DomainInfo1

```
1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
4         DomainInfo();
5         void initialize(const F32 coef_ema=1.0);
6         void setCommInfo(const CommInfo & comm, const PS::F32
              coef_ema=1.0);
7         void setNumberOfDomainMultiDimension(const S32 nx,
8                                               const S32 ny,
9                                               const S32 nz=1);
10        void setDomain(const S32 nx,
11                      const S32 ny,
12                      const S32 nz=1);
13        void setBoundaryCondition(enum BOUNDARY_CONDITION bc);
14        S32 getBoundaryCondition();
15        void setPosRootDomain(const F32vec & low,
16                              const F32vec & high);
17        void setPosRootDomainX(const F64 l, const F64 h);
18        void setPosRootDomainY(const F64 l, const F64 h);
19        void setPosRootDomainZ(const F64 l, const F64 h);
20
21    };
22 }
```

9.1.2.2.1.1 コンストラクタ コンストラクタ

```
void PS::DomainInfo::DomainInfo();
```

- 引数
なし
- 返値
なし
- 機能
領域クラスのオブジェクトを生成する。

9.1.2.2.1.2 *PS::DomainInfo::initialize*

PS::DomainInfo::initialize

```
void PS::DomainInfo::initialize(const PS::F32 coef_ema=1.0);
```

- 引数

coef_ema: 入力。 const PS::F32 型。指数移動平均の平滑化係数。デフォルト 1.0

- 返値

なし

- 機能

領域クラスのオブジェクトを初期化し、指数移動平均の平滑化係数を設定する。この係数の許される値は 0 から 1 である。それ以外の値を入れた場合はエラーメッセージを送出しプログラムは終了する。大きくなるほど、最新の粒子分布の情報が領域分割に反映されやすい。1 の場合、最新の粒子分布の情報のみ反映される。0 の場合、最初の粒子分布の情報のみ反映される。1 度は呼ぶ必要がある。過去の粒子分布の情報を領域分割に反映する必要がある理由については、Ishiyama, Fukushima & Makino (2009, Publications of the Astronomical Society of Japan, 61, 1319) を参照のこと。

9.1.2.2.1.3 *PS::DomainInfo::setCommInfo*

PS::DomainInfo::setCommInfo

```
void setCommInfo(const CommInfo & comm, const PS::F32 coef_ema=1.0);
```

- 引数

comm: 入力。CommInfo &型。

- 返値

なし

- 機能

通信に使うコミュニケータを指定する。

9.1.2.2.1.4 *PS::DomainInfo::setNumberOfDomainMultiDimension*

PS::DomainInfo::setNumberOfDomainMultiDimension

```
void PS::DomainInfo::setNumberOfDomainMultiDimension
    (const PS::S32 nx,
     const PS::S32 ny,
     const PS::S32 nz=1);
```

- 引数

nx: 入力。 const PS::S32 型。 x 軸方向のルートドメインの分割数。

ny: 入力。 const PS::S32 型。 y 軸方向のルートドメインの分割数。

nz: 入力。 const PS::S32 型。 z 軸方向のルートドメインの分割数。デフォルト 1。

- 返値

なし

- 機能

計算領域の分割する方法を設定する。nx, ny, nz はそれぞれ x 軸、y 軸、z 軸方向の計算領域の分割数である。呼ばなければ自動的に nx, ny, nz が決まる。呼んだ場合に入力する nx, ny, nz の総積が MPI プロセス数と等しくなければ、FDPS はエラーメッセージを送り、プログラムを止める。

9.1.2.2.1.5 PS::DomainInfo::setDomain

PS::DomainInfo::setDomain

```
void PS::DomainInfo::setDomain(const PS::S32 nx,
                               const PS::S32 ny,
                               const PS::S32 nz=1);
```

- 引数

nx: 入力。 const PS::S32 型。 x 軸方向のルートドメインの分割数。

ny: 入力。 const PS::S32 型。 y 軸方向のルートドメインの分割数。

nz: 入力。 const PS::S32 型。 z 軸方向のルートドメインの分割数。デフォルト 1。

- 返値

なし

- 機能

PS::DomainInfo::setNumberOfDomainMultiDimension の別名であり、全く同じ動作をする。

9.1.2.2.1.6 *PS::DomainInfo::setBoundaryCondition*

PS::DomainInfo::setBoundaryCondition

```
void PS::DomainInfo::setBoundaryCondition  
    (enum PS::BOUNDARY_CONDITION bc);
```

- 引数

bc: 入力。 列挙型。境界条件。

- 返値

なし

- 機能

境界条件の設定をする。許される入力は、6.8.2 で挙げた列挙型のみ (ただし BOUNDARY_CONDITION_SHEARING_BOX, BOUNDARY_CONDITION_USER_DEFINED は未実装)。呼ばない場合は、開放境界となる。

9.1.2.2.1.7 *PS::DomainInfo::getBoundaryCondition*

PS::DomainInfo::getBoundaryCondition

```
S32 PS::DomainInfo::getBoundaryCondition();
```

- 引数

なし

- 返値

PS::S32 型。

- 機能

設定されている境界条件に対応する整数を返す。取りうる値はPS::BOUNDARY_CONDITION 型の各列挙子に割り当てられている整数のいずれかである。

9.1.2.2.1.8 *PS::DomainInfo::setPosRootDomain*

PS::DomainInfo::setPosRootDomain

```
void PS::DomainInfo::setPosRootDomain  
    (const PS::F32vec & low,  
     const PS::F32vec & high);
```

- 引数

low: 入力。 PS::F32vec 型。計算領域の下限 (閉境界)。

high: 入力。 PS::F32vec 型。計算領域の上限 (開境界)。

- 返値

なし

- 機能

計算領域の下限と上限を設定する。開放境界条件の場合は呼ぶ必要はない。それ以外の境界条件の場合は、呼ばなくても動作するが、その結果が正しいことは保証できない。high の座標の各値は low の対応する座標よりも大きくなければならない。そうでない場合は、FDPS はエラーメッセージを送出し、ユーザープログラムを終了させる。

9.1.2.2.1.9 PS::DomainInfo::setPosRootDomainX

PS::DomainInfo::setPosRootDomainX

```
void PS::DomainInfo::setPosRootDomainX
    (const F64 l, const F64 h);
```

- 引数

l: 入力。 PS::F64 型。計算領域の x 座標の下限 (閉境界)。

h: 入力。 PS::F64c 型。計算領域の x 座標の上限 (開境界)。

- 返値

なし

- 機能

計算領域の x 座標の下限と上限を設定する。開放境界条件の場合は通常呼ぶ必要はない。それ以外の境界条件の場合は、呼ばなくても動作するが、その結果が正しいことは保証できない。開放境界の場合でも、TreeForce に与える CALC_DISTANCE_TYPE 型が NORNAL ではない時には想定する条件に応じた上限・下限をこの関数で設定し、実際にの座標値がその範囲に収まる必要がある。

high の座標の各値は low の対応する座標よりも大きくなければならない。そうでない場合は、FDPS はエラーメッセージを送出し、ユーザープログラムを終了させる。

9.1.2.2.1.10 PS::DomainInfo::setPosRootDomainY

PS::DomainInfo::setPosRootDomainY

```
void PS::DomainInfo::setPosRootDomainY
    (const F64 l, const F64 h);
```

setPosRootDomainX と同様だが y 座標についての関数である。

9.1.2.2.1.11 PS::DomainInfo::setPosRootDomainZ

PS::DomainInfo::setPosRootDomainZ

```
void PS::DomainInfo::setPosRootDomainZ
    (const F64 l, const F64 h);
```

setPosRootDomainX と同様だが z 座標についての関数である。

9.1.2.2.2 領域分割

領域分割関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 29: DomainInfo2

```
1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
4         template<class Tpsys>
5         void collectSampleParticle(Tpsys & psys,
6                                     const bool clear,
7                                     const F32 weight);
8         template<class Tpsys>
9         void collectSampleParticle(Tpsys & psys,
10                                    const bool clear);
11        template<class Tpsys>
12        void collectSampleParticle(Tpsys & psys);
13
14        void decomposeDomain();
15
16        template<class Tpsys>
17        void decomposeDomainAll(Tpsys & psys,
18                                const F32 weight);
19        template<class Tpsys>
20        void decomposeDomainAll(Tpsys & psys);
21    };
```

9.1.2.2.1 PS::DomainInfo::collectSampleParticle

PS::DomainInfo::collectSampleParticle

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
    (Tpsys & psys,
     const bool clear,
     const PS::F32 weight);
```

- 引数

psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

clear: 入力。 bool 型。前にサンプルされた粒子情報をクリアするかどうかを決定するフラグ。 true でクリアする。

weight: 入力。 const PS::F32 型。領域分割のためのサンプル粒子数を決めるためのウェイト。

- 返値

なし

- 機能

粒子群クラスのオブジェクト psys から粒子をサンプルする。clear によってこれより前にサンプルした粒子の情報を消すかどうか決める。weight によってその MPI プロセスからサンプルする粒子の量を調整する (weight が大きいほどサンプル粒子数が多い)。具体的には、プロセス i の weight を w_i 、粒子群クラスの API

setAverageTargetNumberOfSampleParticlePerProcess

で設定されたプロセスあたりのサンプル粒子数を n_{smp} 、プロセス数を n_{proc} とすると、プロセス i からは $n_{\text{smp}} n_{\text{proc}} (w_i / \sum_k w_k)$ 個の粒子数がサンプリングされる。通常、weight にはそのプロセスでの計算時間を反映する量を指定する。

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
    (Tpsys & psys,
     const bool clear);
```

- 引数

psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

clear: 入力。 bool 型。前にサンプルされた粒子情報をクリアするかどうかを決定するフラグ。 true でクリアする。

- 返値

なし

- 機能

粒子群クラスのオブジェクト psys から粒子をサンプルする。 clear によってこれより前にサンプルした粒子の情報を消すかどうか決める。

```
template<class Tpsys>
void PS::DomainInfo::collectSampleParticle
    (Tpsys & psys);
```

- 引数

psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

- 返値

なし

- 機能

粒子群クラスのオブジェクト psys から粒子をサンプルする。

9.1.2.2.2.2 PS::DomainInfo::decomposeDomain

PS::DomainInfo::decomposeDomain

```
void PS::DomainInfo::decomposeDomain();
```

- 引数

なし

- 返値

なし

- 機能

計算領域の分割を実行する。

9.1.2.2.2.3 PS::DomainInfo::decomposeDomainAll

PS::DomainInfo::decomposeDomainAll

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomainAll
    (Tpsys & psys,
     const PS::F32 weight);
```

- 引数

psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラス。

weight: 入力。 const PS::F32 型。領域分割のためのサンプル粒子数を決めるためのウェイト。

- 返値

なし

- 機能

粒子群クラスのオブジェクト psys から粒子をサンプルし、続けてルートドメインの分割を行う。PS::DomainInfo::collectSampleParticle と PS::DomainInfo::decomposeDomain が行うことを一度に行う。weight の意味は PS::DomainInfo::collectSampleParticle と同じ。

```
template<class Tpsys>
void PS::DomainInfo::decomposeDomainAll
    (Tpsys & psys);
```

- 引数

psys: 入力。 ParticleSystem &型。領域分割のためのサンプル粒子を提供する粒子群クラスのオブジェクト。

- 返値

なし

- 機能

粒子群クラスのオブジェクト psys から粒子をサンプルし、続けてルートドメインの分割を行う。PS::DomainInfo::collectSampleParticle と PS::DomainInfo::decomposeDomain が行うことを一度に行う。

9.1.2.2.3 時間計測

クラス内の時間計測関連の API の宣言は以下のようにになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバの `time_profile_` の該当メンバに書き込む。メソッド `clearTimeProfile()` を呼ばない限り時間は足しあわされていく。

ソースコード 30: DomainInfo3

```
1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
4         TimeProfile getTimeProfile();
5         void clearTimeProfile();
6     };
7 }
```

9.1.2.2.3.1 *PS::DomainInfo::getTimeProfile*

PS::DomainInfo::getTimeProfile

```
PS::TimeProfile PS::DomainInfo::getTimeProfile();
```

- 引数

なし。

- 返値

PS::TimeProfile 型。

- 機能

メンバ関数 `collectSampleParticle` と `decomposeDomain` にかかった時間（ミリ秒単位）を `TimeProfile` 型のメンバ変数 `collect_sample_particles_` と `decompose_domain_` に格納する。

9.1.2.2.3.2 *PS::DomainInfo::clearTimeProfile*

PS::DomainInfo::clearTimeProfile

```
void PS::DomainInfo::clearTimeProfile();
```

- 引数

なし。

- 返値

なし。

- 機能

領域情報クラスの TimeProfile 型のプライベートメンバ変数のメンバ変数 collect_sample_particles_ と decompose_domain_ の値を 0 クリアする。

9.1.2.2.4 情報取得

クラス内の情報取得関連の API の宣言は以下のようになっている。

ソースコード 31: DomainInfo3

```

1 namespace ParticleSimulator {
2     class DomainInfo{
3     public:
4         TimeProfile getTimeProfile();
5         void clearTimeProfile();
6         S64 getUsedMemorySize();
7     };
8 }
```

9.1.2.2.4.1 PS::DomainInfo::getUsedMemorySize

PS::DomainInfo::getUsedMemorySize

PS::S64 PS::DomainInfo::getUsedMemorySize();

- 引数

なし。

- 返値

PS::S64。

- 機能

対象のオブジェクトが使用しているメモリー量を Byte 単位で返す。

9.1.3 粒子群クラス

本節では、粒子群クラスについて記述する。このクラスは粒子情報の保持や MPI プロセス間で粒子情報の交換を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.3.1 オブジェクトの生成

粒子群クラスは以下のように宣言されている。

ソースコード 32: ParticleSystem0

```
1 namespace ParticleSimulator {  
2     template<class Tptcl>  
3     class ParticleSystem;  
4 }
```

テンプレート引数 Tptcl はユーザー定義の FullParticle クラスである。

粒子群クラスのオブジェクトの生成は以下のように行う。ここでは system というオブジェクトを生成している。

```
PS::ParticleSystem<FP> system;
```

テンプレート引数 FP はユーザー定義の FullParticle クラスの 1 例である FP クラスである。

9.1.3.2 API

このモジュールには初期設定関連の API、オブジェクト情報取得設定関連の API、ファイル入出力関連の API、粒子交換関連の API がある。以下、各節に分けて記述する。

9.1.3.2.1 初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 33: ParticleSystem1

```
1 namespace ParticleSimulator {  
2     template<class Tptcl>  
3     class ParticleSystem{  
4     public:  
5         ParticleSystem();  
6         void initialize();  
7         void setCommInfo(const CommInfo & comm);  
8         void setAverageTargetNumberOfSampleParticlePerProcess  
9             (const S32 & nsampleperprocess);  
10    };  
11 }
```

9.1.3.2.1.1 コンストラクタ

コンストラクタ

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::ParticleSystem();
```

- 引数
なし
- 返値
なし
- 機能
粒子群クラスのオブジェクトを生成する。

9.1.3.2.1.2 *PS::ParticleSystem::initialize*

PS::ParticleSystem::initialize

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::initialize();
```

- 引数
なし
- 返値
なし
- 機能
粒子群クラスのオブジェクトを初期化する。1度は呼ぶ必要がある。

9.1.3.2.1.3 *PS::ParticleSystem::setCommInfo*

PS::ParticleSystem::setCommInfo

```
void setCommInfo(const CommInfo & comm);
```

- 引数
comm: 入力 . CommInfo &型 .

- 返値
なし
- 機能
通信に使うコミュニケータを指定する。

9.1.3.2.1.4 PS::ParticleSystem::

setAverateTargetNumberOfSampleParticlePerProcess

PS::ParticleSystem::

setAverateTargetNumberOfSampleParticlePerProcess

```
template <class Tptcl>
void PS::ParticleSystem<Tptcl>::setAverateTargetNumberOfSampleParticlePerProcess
    (const PS::S32 & nsampleperprocess);
```

- 引数
nsampleperprocess: 入力。const PS::S32 &型。1つのMPIプロセスでサンプルする粒子数目標。
- 返値
なし
- 機能
1つのMPIプロセスでサンプルする粒子数の目標を設定する。呼び出さなくてもよいが、呼び出さないとこの目標数が30となる。

9.1.3.2.2 情報取得

オブジェクト情報取得関連のAPIの宣言は以下のようにになっている。このあと各APIについて記述する。

ソースコード 34: ParticleSystem2

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem{
4     public:
5         Tptcl & operator [] (const S32 id);
6         S32 getNumberOfParticleLocal() const;
7         S32 getNumberOfParticleGlobal() const;
8         S64 getUsedMemorySize() const;
```

```
9     };  
10 }
```

9.1.3.2.2.1 *PS::ParticleSystem::operator []*
PS::ParticleSystem::operator []

```
template <class Tptcl>  
Tptcl & PS::ParticleSystem<Tptcl>::operator []  
    (const PS::S32 id);
```

- 引数
id: 入力。const PS::S32 型。粒子配列のインデックス。
- 返値
FullParticle &型。Tptcl 型のオブジェクト。
- 機能
id 番目の FullParticle 型のオブジェクトの参照を返す。

9.1.3.2.2.2 *PS::ParticleSystem::getNumberOfParticleLocal*
PS::ParticleSystem::getNumberOfParticleLocal

```
template <class Tptcl>  
PS::S32 PS::ParticleSystem<Tptcl>::getNumberOfParticleLocal();
```

- 引数
なし
- 返値
PS::S32 型。1 つの MPI プロセスの持つ粒子数。
- 機能
1 つの MPI プロセスの持つ粒子数を返す。

9.1.3.2.2.3 *PS::ParticleSystem::getNumberOfParticleGlobal*

PS::ParticleSystem::getNumberOfParticleGlobal

```
template <class Tptcl>
PS::S32 PS::ParticleSystem<Tptcl>::getNumberOfParticleGlobal();
```

- 引数
なし
- 返値
PS::S32 型。全 MPI プロセスの持つ粒子数。
- 機能
全 MPI プロセスの持つ粒子数を返す。

9.1.3.2.2.4 *PS::DomainInfo::getUsedMemorySize*

PS::DomainInfo::getUsedMemorySize

```
PS::S64 PS::DomainInfo::getUsedMemorySize() const;
```

- 引数
なし。
- 返値
PS::S64。
- 機能
対象のオブジェクトが使用しているメモリー量を Byte 単位で返す。

9.1.3.2.3 ファイル入出力

ファイル入出力関連の API は `void (read|write)Paritlce(Ascii|Binary)` であり、プレフィックスの `read` は入力、`write` は出力、サフィックスの `Ascii` はファイルがアスキー形式の場合、`Binary` はファイルがバイナリ形式の場合に使用する。それぞれの関数に対してさらに、引数が異なる 8 つの関数がオーバーロードされている。それらを下記に示す。

9.1.3.2.3.1 PS::ParticleSystem::readParticleAscii

PS::ParticleSystem::readParticleAscii

readParticleAscii は下記の 8 つの異なる引数仕様を持つ。

```
1 void PS::ParticleSystem<Tptlc>::readParticleAscii
2     (Tchar0 filename,
3      Tchar1 format,
4      Theader & header);
5
6 void PS::ParticleSystem<Tptcl>::readParticleAscii
7     (Tchar0 filename,
8      Tchar1 format);
9
10 void PS::ParticleSystem<Tptcl>::readParticleAscii
11     (Tchar0 filename,
12      Theader & header);
13
14 void PS::ParticleSystem<Tptcl>::readParticleAscii
15     (Tchar0 filename);
16
17 void PS::ParticleSystem<Tptlc>::readParticleAscii
18     (Tchar0 filename,
19      Tchar1 format,
20      Theader & header,
21      void (Tptcl::*pFunc)(FILE*));
22
23 void PS::ParticleSystem<Tptcl>::readParticleAscii
24     (Tchar0 filename,
25      Tchar1 format,
26      void (Tptcl::*pFunc)(FILE*));
27
28 void PS::ParticleSystem<Tptcl>::readParticleAscii
29     (Tchar0 filename,
30      Theader & header,
31      void (Tptcl::*pFunc)(FILE*));
32
33 void PS::ParticleSystem<Tptcl>::readParticleAscii
34     (Tchar0 filename,
35      void (Tptcl::*pFunc)(FILE*));
```

- 引数

引数の型 `Tcha0` と `Tchar1` は `const char * const`, `const char *`, もしくは `char *` 型を表す。

`filename`: 入力。 `const char * const`, `const char *`, もしくは `char *` 型。それ以外の型が与えられた場合の動作は未定義。分散ファイル読み込み時には入力ファイル名のベースとなる部分。単一ファイル読み込み時には入力ファイル名。

`format`: 入力。 `const char * const`, `const char *`, もしくは `char *` 型。分散ファイルから粒子データを読み込む際のファイルフォーマット。

`header`: 入力。 `Theader &` 型。ファイルのヘッダ情報。

`pFunc`: 入力。 `void (Tptcl::*)(FILE*)` 型。 `FILE` ポインタを引数にとり `void` を返す `Tptcl` のメンバ関数ポインタ。

- 返値

なし

- 機能

引数 `format` が存在する場合、各プロセスが `filename` と `format` の組によって指定される名前の入力ファイルから粒子データを読み出し、データを `FullParticle` クラスのオブジェクトに格納する (分散ファイルの読み込み)。一方、引数 `format` が存在しない場合、ルートプロセスが `filename` で指定された入力ファイルから粒子データを読み出し、データを `FullParticle` クラスのオブジェクトに格納した後、各プロセスに分配する (単一ファイルの読み込み)。いずれの場合においても、引数 `header` が存在しない場合、ファイルに保存されている粒子数を調べるため、1 回ファイルを読み込んで行数を取得した後、もう一度ファイルを読み込み直し、粒子データを読み込む。すなわち、粒子数の推定は、1 粒子のデータが 1 行に記述されているという仮定の下行う。

分散ファイルを読み込む場合、`filename` で、分散しているファイルのベースとなる名前を指定する。`format` でファイル名のフォーマットを指定する。フォーマットの指定方法は標準 C ライブラリの関数 `printf` の第 1 引数と同じである。ただし変換指定は必ず 3 つであり、その指定子は 1 つめは文字列、残りはどちらも整数である。2 つ目の変換指定にはそのジョブの全プロセス数が、3 つ目の変換指定にはプロセス番号が入る。例えば、`filename` が `nbody`、`format` が `%s_%03d_%03d.init` ならば、全プロセス数 64 のジョブのプロセス番号 12 のプロセスは、`nbody_064_012.init` というファイルを読み込む。

1 粒子のデータを読み取る関数は引数 `pFunc` が存在すればそれを使用し、そうでない場合には `FullParticle` クラスのメンバ関数 `readAscii` を使用する。`readAscii` はユーザが定義しなければならない。`readAscii` の仕様については節 7.2.4.3.1 を、実装例については節 A.1.4.3 を参照のこと。`pFunc` も `readAscii` と同じ仕様を満たす必要がある。

ファイルのヘッダのデータを読み取る関数は `Theader` のメンバ関数 `readAscii` でユーザが定義する。仕様については節 7.8.3.1 を、実装例については節 A.7 を参照のこと。

ファイルはアスキーモードで開く。

9.1.3.2.3.2 *PS::ParticleSystem::readParticleBinary*

`PS::ParticleSystem::readParticleBinary`

`readParticleBinary` は下記の 8 つの異なる引数仕様を持つ。

```
1 void PS::ParticleSystem<Tptlc>::readParticleBinary
2     (Tchar0 filename,
3      Tchar1 format,
4      Theader & header);
5
6 void PS::ParticleSystem<Tptcl>::readParticleBinary
7     (Tchar0 filename,
8      const char * const format);
9
10 void PS::ParticleSystem<Tptcl>::readParticleBinary
11     (Tchar0 filename,
12      Theader & header);
13
14 void PS::ParticleSystem<Tptcl>::readParticleBinary
15     (Tchar0 filename);
16
17 void PS::ParticleSystem<Tptlc>::readParticleBinary
18     (Tchar0 filename,
19      Tchar1 format,
20      Theader & header,
21      void (Tptcl::*pFunc)(FILE*));
22
23 void PS::ParticleSystem<Tptcl>::readParticleBinary
24     (Tchar0 filename,
25      Tchar1 format,
26      void (Tptcl::*pFunc)(FILE*));
27
28 void PS::ParticleSystem<Tptcl>::readParticleBinary
29     (Tchar0 filename,
30      Theader & header,
31      void (Tptcl::*pFunc)(FILE*));
32
33 void PS::ParticleSystem<Tptcl>::readParticleBinary
34     (Tchar0 filename,
35      void (Tptcl::*pFunc)(FILE*));
36
37 void PS::ParticleSystem<Tptcl>::writeParticleAscii
```

```

38         (Tchar0 const filename,
39         const Theader & header,
40         void (Tptcl::*pFunc)(FILE*)const);

```

- 引数

引数の型 Tcha0 と Tchar1 は const char * const, const char *, もしくは char * 型を表す。

filename: 入力。const char *型。分散ファイル読み込み時には入力ファイル名のベースとなる部分。単一ファイル読み込み時にはファイル名。

format: 入力。const char *型。分散ファイルから粒子データを読み込む際のファイルフォーマット。

header: 入力。Theader &型。ファイルのヘッダ情報。

pFunc: 入力。void (Tptcl::*)(FILE*) 型。FILE ポインタを引数にとり void を返す Tptcl のメンバ関数ポインタ。

- 返値

なし

- 機能

readParticleAscii とほぼ同じ機能を提供するが、下記の点が異なる。

- ファイルはバイナリーモードで開く。
- 引数 header が存在しない場合の粒子数の推定は、実際に 1 粒子を読み込んだときのファイル位置指示子のバイト数を調べることによって行うため、1 粒子のデータの最後が改行文字 (\n) となっている必要はない。

9.1.3.2.3.3 PS::ParticleSystem::writeParticleAscii

PS::ParticleSystem::writeParticleAscii

writeParticleAscii は下記の 8 つの異なる引数仕様を持つ。

```

1 template <class Tptcl>
2 template <class Theader>
3 void PS::ParticleSystem<Tptcl>::writeParticleAscii
4     (const char * const filename,
5     const char * const format,
6     const Theader & header);
7
8 template <class Tptcl>
9 void PS::ParticleSystem<Tptcl>::writeParticleAscii

```

```

10         (const char * const filename,
11         const char * const format);
12
13 template <class Tptcl>
14 template <class Theader>
15 void PS::ParticleSystem<Tptcl>::writeParticleAscii
16         (const char * const filename,
17         const Theader & header);
18
19 template <class Tptcl>
20 void PS::ParticleSystem<Tptcl>::writeParticleAscii
21         (const char * const filename);
22
23 template <class Tptcl>
24 template <class Theader>
25 void PS::ParticleSystem<Tptcl>::writeParticleAscii
26         (const char * const filename,
27         const char * const format,
28         const Theader & header,
29         void (Tptcl::*pFunc)(FILE*)const);
30
31 template <class Tptcl>
32 void PS::ParticleSystem<Tptcl>::writeParticleAscii
33         (const char * const filename,
34         const char * const format,
35         void (Tptcl::*pFunc)(FILE*)const);
36
37 template <class Tptcl>
38 template <class Theader>
39 void PS::ParticleSystem<Tptcl>::writeParticleAscii
40         (const char * const filename,
41         const Theader & header,
42         void (Tptcl::*pFunc)(FILE*)const);
43
44 template <class Tptcl>
45 void PS::ParticleSystem<Tptcl>::writeParticleAscii
46         (const char * const filename,
47         void (Tptcl::*pFunc)(FILE*)const);

```

- 引数

filename: 入力。const char * const 型。出力ファイル名のベースとなる部分。

format: 入力。const char * const 型。分散ファイルに粒子データを書き込む際のファイルフォーマット。

header: 入力。const Theader &型。ファイルのヘッダ情報。

pFunc: 入力。void (Tptcl::*)(FILE*)const 型。FILE ポインタを引数にとり void を返す Tptcl のメンバ関数ポインタ。

- 返り値

なし。

- 機能

引数 format が存在する場合、各プロセスが filename と format の組によって指定される名前の出力ファイルに FullParticle クラスのオブジェクトのデータと、もし存在すれば Theader クラスのオブジェクトのデータを出力する (分散ファイルへの書き出し)。一方、引数 format が存在しない場合、各プロセスの FullParticle クラスのオブジェクトのデータをルートプロセスに集め、ルートプロセスが filename で指定された出力ファイルに集めたデータと、もし存在すれば Theader クラスのオブジェクトデータを出力する (単一ファイルへの書き出し)。

分散ファイル書き出し時の出力ファイル名のフォーマットはメンバ関数

PS::ParticleSystem::readParticleAscii の場合と同様である。

1 粒子のデータを書き込む関数は pFunc が存在する場合にはそれが使用され、そうでない場合には FullParticle クラスのメンバ関数 writeAscii が使用される。writeAscii はユーザが定義しなければならない。writeAscii の仕様については節 7.2.4.3.2 を、実装例については節 A.1.4.3 を参照のこと。

ファイルのヘッダのデータを書き込む関数はヘッダクラスのメンバ関数 writeAscii でユーザが定義する。仕様については節 7.8.3.2 を、実装例については節 A.7 を参照のこと。

ファイルはアスキーモードで開く。

9.1.3.2.3.4 PS::ParticleSystem::writeParticleBinary

PS::ParticleSystem::writeParticleBinary

writeParticleBinary は下記の 8 つの異なる引数仕様を持つ。

```
1 template <class Tptcl>
2 template <class Theader>
3 void PS::ParticleSystem<Tptcl>::writeParticleBinary
4         (const char * const filename,
5         const char * const format,
```

```

6         const Theader & header);
7
8 template <class Tptcl>
9 void PS::ParticleSystem<Tptcl>::writeParticleBinary
10         (const char * const filename,
11         const char * const format);
12
13 template <class Tptcl>
14 template <class Theader>
15 void PS::ParticleSystem<Tptcl>::writeParticleBinary
16         (const char * const filename,
17         const Theader & header);
18
19 template <class Tptcl>
20 void PS::ParticleSystem<Tptcl>::writeParticleBinary
21         (const char * const filename);
22
23 template <class Tptcl>
24 template <class Theader>
25 void PS::ParticleSystem<Tptcl>::writeParticleBinary
26         (const char * const filename,
27         const char * const format,
28         const Theader & header,
29         void (Tptcl::*pFunc)(FILE*)const);
30
31 template <class Tptcl>
32 void PS::ParticleSystem<Tptcl>::writeParticleBinary
33         (const char * const filename,
34         const char * const format,
35         void (Tptcl::*pFunc)(FILE*)const);
36
37 template <class Tptcl>
38 template <class Theader>
39 void PS::ParticleSystem<Tptcl>::writeParticleBinary
40         (const char * const filename,
41         const Theader & header,
42         void (Tptcl::*pFunc)(FILE*)const);
43
44 template <class Tptcl>
45 void PS::ParticleSystem<Tptcl>::writeParticleBinary

```

```
46         (const char * const filename ,  
47         void (Tptcl::*pFunc)(FILE*)const));
```

- 引数

filename: 入力。const char * const 型。分散ファイルへの書き込み時には出力ファイル名のベースとなる部分。単一ファイルへの書き込み時にはファイル名。

format: 入力。const char * const 型。分散ファイルに粒子データを書き込む際のファイルフォーマット。

header: 入力。const Theader &型。ファイルのヘッダ情報。

pFunc: 入力。void (Tptcl::*)(FILE*)const 型。FILE ポインタを引数にとり void を返す Tptcl のメンバ関数ポインタ。

- 返り値

なし。

- 機能

writeParticleAscii とほぼ同じ機能を提供するが、下記の点が異なる。

- ファイルはバイナリーモードで開く。

9.1.3.2.4 粒子交換

粒子交換関連の API の宣言は以下のようにになっている。このあと各 API について記述する。

ソースコード 35: ParticleSystem4

```
1 namespace ParticleSimulator {  
2     template<class Tptcl>  
3     class ParticleSystem{  
4     public:  
5         template<class Tdinfo>  
6         void exchangeParticle(Tdinfo & dinfo, const bool  
           flag_serialize=false);  
7     };  
8 }
```

9.1.3.2.4.1 PS::ParticleSystem::exchangeParticle

PS::ParticleSystem::exchangeParticle

```
template <class Tptcl>
template <class Tdinfo>
void PS::ParticleSystem<Tptcl>::exchangeParticle
    (Tdinfo & dinfo, const bool flag_serialize=false);
```

- 引数

dinfo: 入力。DomainInfo & 型。領域クラスのオブジェクト。

flag_serialize: 入力。const bool 型。粒子情報をシリアル化して送信するかを決定するフラグ。true で粒子をシリアル化する。デフォルトは false。

- 返値

なし

- 機能

粒子が適切なドメインに配置されるように、粒子の交換を行う。粒子をシリアル化して送信する場合は FP にメンバ関数 pack と unPack を定義し (詳しくはセクション 7.2.4.6)、flag_serialize を true にする (**シリアル化送信の対応は未実装**)。

9.1.3.2.5 粒子の追加、削除

粒子の追加もしくは削除関連の API は以下の様に宣言されている。

9.1.3.2.5.1 PS::ParticleSystem::addOneParticle()

PS::ParticleSystem::addOneParticle()

```
void PS::ParticleSystem::addOneParticle(const FullParticle & fp);
```

- 引数

fp: 入力。const FullParticle &型。追加する粒子の FullParticle。

- 返値

なし。

- 機能

追加された粒子を粒子配列の末尾に追加する。

9.1.3.2.5.2 *PS::ParticleSystem::removeParticle()*

PS::ParticleSystem::removeParticle()

```
void PS::ParticleSystem::removeParticle(const PS::S32 * idx,  
                                         const PS::S32 n);
```

- 引数

idx: 入力。const PS::S32 * 型。消去する粒子の配列インデックスの配列。

n: 入力。const PS::S32 型。配列 idx のサイズ。

- 返値

なし。

- 機能

配列 idx[] に格納されているインデックスの粒子を削除する。この関数を呼ぶ前後で、粒子の配列インデックスが同じである事は保証されない。

9.1.3.2.6 時間計測

クラス内の情報取得関連の API の宣言は以下のようにになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバの time_profile_ の該当メンバに書き込む。メソッド clearTimeProfile() を呼ばない限り時間は足しあわされていく。

ソースコード 36: ParticleSystem3

```
1 namespace ParticleSimulator {  
2     template<class Tptcl>  
3     class ParticleSystem{  
4     private:  
5         TimeProfile time_profile_;  
6     public:  
7         TimeProfile getTimeProfile();  
8         void clearTimeProfile();  
9     };  
10 }
```

9.1.3.2.6.1 *PS::ParticleSystem::getTimeProfile*

PS::ParticleSystem::getTimeProfile

```
PS::TimeProfile PS::ParticleSystem::getTimeProfile();
```

- 引数
なし。
- 返値
PS::TimeProfile 型。
- 機能
メンバ関数 exchangeParticle にかかった時間（ミリ秒単位）を TimeProfile 型のメンバ変数 exchange_particles_ に格納し、返す。

9.1.3.2.6.2 PS::ParticleSystem::clearTimeProfile

PS::ParticleSystem::clearTimeProfile

```
void PS::ParticleSystem::clearTimeProfile();
```

- 引数
なし。
- 返値
なし。
- 機能
領域情報クラスの TimeProfile 型のプライベートメンバ変数のメンバ変数 exchange_particles_ の値を 0 クリアする。

9.1.3.2.7 その他

その他の API の宣言は以下のようにになっている。このあと各 API について記述する。

ソースコード 37: ParticleSystem4

```
1 namespace ParticleSimulator {
2     template<class Tptcl>
3     class ParticleSystem{
4     public:
5         template<class Tdinfo>
6         void adjustPositionIntoRootDomain
7             (const Tdinfo & dinfo);
8         void setNumberOfParticleLocal(const S32 n);
9         template<class Tcomp>
10        void sortParticle(Tcomp comp);
```

```
11     };  
12 }
```

9.1.3.2.7.1 *PS::ParticleSystem::adjustPositionIntoRootDomain*

PS::ParticleSystem::adjustPositionIntoRootDomain

```
template <class Tptcl>  
template <class Tdinfo>  
void ParticleSystem<Tptcl>::adjustPositionIntoRootDomain  
    (const Tdinfo & dinfo);
```

- 引数
dinfo: 入力。Tdinfo 型。領域クラスのオブジェクト。
- 返値
なし
- 機能
周期境界条件の場合に、計算領域からはみ出した粒子を計算領域に適切に戻す。

9.1.3.2.7.2 *PS::ParticleSystem::setNumberOfParticleLocal*

PS::ParticleSystem::setNumberOfParticleLocal

```
template <class Tptcl>  
void PS::ParticleSystem<Tptcl>::setNumberOfParticleLocal  
    (const PS::S32 n);
```

- 引数
n: 入力。const PS::S32 型。粒子数。
- 返値
なし
- 機能
1 つの MPI プロセスの持つ粒子数を設定する。

9.1.3.2.7.3 PS::ParticleSystem::sortParticle

PS::ParticleSystem::sortParticle

```
template<class Tcomp>
void sortParticle(Tcomp comp);
```

- 引数

comp: 入力。Tcomp 型。比較関数。比較関数は戻り値を bool 型とし、引数は const FullParticle & を 2 つ。例として以下に FullParticle がメンバ変数 id を持っており id の昇順に並べ替える場合を示す。

```
bool comp(const FP & left, const FP & right){
    return left.id < right.id;
}
```

- 返値

なし

- 機能

粒子群クラスが保持する FullParticle の配列を引数 comp で指示したように並べ替える。

9.1.4 相互作用ツリークラス

本節では、相互作用ツリークラスについて記述する。このクラスは粒子間相互作用の計算を行うモジュールである。まずオブジェクトの生成方法を記述し、その後 API を記述する。

9.1.4.1 オブジェクトの生成

このクラスは以下のように宣言されている。

ソースコード 38: TreeForForce0

```
1 namespace ParticleSimulator {
2     template<class TSearchMode,
3             class TResult,
4             class TEpi,
5             class TEpj,
6             class TMomLocal,
7             class TMomGlobal,
8             class TSpj,
```



```

9          enum CALC_DISTANCE_TYPE CALC_DISTANCE_TYPE =
              CALC_DISTANCE_TYPE_NORMAL >
10      class TreeForForce;
11  }

```

テンプレート引数は順に、PS::SEARCH_MODE 型 (ユーザー選択)、Force クラス (ユーザー定義)、EssentialParticleI クラス (ユーザー定義)、EssentialParticleJ 型 (ユーザー定義)、ローカルツリーの Moment 型 (ユーザー定義)、グローバルツリーの Moment 型 (ユーザー定義)、SuperParticleJ 型 (ユーザー定義)、PS::CALC_DISTANCE_TYPE 型は粒子間の距離の計算方法を決定する為に FDPS で定義されている型である。(詳細は 6.8.5 節を参照)

PS::SEARCH_MODE 型に応じてラッパーを用意した。これらのラッパーを使えば入力するテンプレート引数の数が減るので、こちらのラッパーを用いることを推奨する。以下、各 PS::SEARCH_MODE 型の場合のオブジェクトの生成方法を記述する。

9.1.4.1.1 PS::SEARCH_MODE_LONG

以下のようにオブジェクト system を生成する。

```

PS::TreeForForceLong
    <TResult, TEpi, TEpj, TMom, TSpj>::Normal system;

```

テンプレート引数は順に、Force クラス (ユーザー定義)、EssentialParticleI クラス (ユーザー定義)、EssentialParticleJ クラス (ユーザー定義)、ローカルツリー及びグローバルツリーの Moment クラス (ユーザー定義)、SuperParticleJ クラス (ユーザー定義) である。

Moment クラスと SuperParticleJ クラスに FDPS が提供する Moment クラスと SuperParticleJ クラスを指定した型も用意した。これらはモーメントの計算方法別に 6 種類ある。以下、粒子の重心を中心とした場合の単極子まで、四重極子までのモーメント計算、粒子の幾何中心を中心とした場合の単極子まで、双極子まで、四重極子までのモーメント計算、のオブジェクトの生成方法をこの順で記述する。ここでは、すべて system というオブジェクトを生成している。

```

PS::TreeForForceLong
    <TResult, TEpi, TEpj>::Monopole system;

```

```

PS::TreeForForceLong
    <TResult, TEpi, TEpj>::Quadrupole system;

```

```

PS::TreeForForceLong
    <TResult, TEpi, TEpj>::MonopoleGeometricCenter system;

```

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj>::DipoleGeometricCenter system;
```

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj>::QuadrupoleGeometricCenter system;
```

各相互作用ツリークラスで使用している Moment クラスと SuperParticleJ クラスは、上記の順で、

- PS::MomentMonopole と PS::SPJMonopole (定義は 7.5.2.2.1, 7.6.2.1.1 参照)
- PS::MomentQuadrupole と PS::SPJQuadrupole (定義は 7.5.2.2.2, 7.6.2.1.2 参照)
- PS::MomentMonopoleGeometricCenter と PS::SPJMonopoleGeometricCenter (定義は 7.5.2.2.3, 7.6.2.1.3 参照)
- PS::MomentDipoleGeometricCenter と PS::SPJDipoleGeometricCenter (定義は 7.5.2.2.4, 7.6.2.1.4 参照)
- PS::MomentQuadrupoleGeometricCenter と PS::SPJQuadrupoleGeometricCenter (定義は 7.5.2.2.5, 7.6.2.1.5 参照)

である。

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.2 PS::SEARCH_MODE_LONG_SCATTER

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj, TMom, TSpj>::WithScatterSearch system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラス、ローカルツリー及びグローバルツリーの Moment クラス、SuperParticleJ クラスである。

Moment クラスと SuperParticleJ クラスに FDPS が提供する Moment クラスと SuperParticle J クラスを指定したクラスも用意した。これらはモーメントの計算方法別に 2 種類ある。以下、粒子の重心を中心とした場合の単極子まで、四重極子までのモーメント計算のオブジェクトの生成方法をこの順で記述する。ここでは、すべて system というオブジェクトを生成している。

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj>::MonopoleWithScatterSearch system;
```

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj>::QuadrupoleWithScatterSearch system;
```

各相互作用ツリークラスで使用している Moment クラスと SuperParticleJ クラスは、上記の順で、PS::MomentMonopoleScatter と PS::SPJMonopoleScatter (定義は 7.5.2.3.1, 7.6.2.2.1 参照)、PS::MomentQuadrupoleScatter と PS::SPJQuadrupoleScatter (定義は 7.5.2.3.2, 7.6.2.2.2 参照) である。

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.3 PS::SEARCH_MODE_LONG_SYMMETRY

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj, TMom, TSpj>::WithSymmetrySearch system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラス、ローカルツリー及びグローバルツリーの Moment クラス、SuperParticleJ クラスである。

Moment クラスと SuperParticleJ クラスに FDPS が提供する Moment クラスと SuperParticleJ クラスを指定したクラスも用意した。これらはモーメントの計算方法別に 2 種類ある。以下、粒子の重心を中心とした場合の単極子まで、四重極子までのモーメント計算のオブジェクトの生成方法をこの順で記述する。ここでは、すべて system というオブジェクトを生成している。

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj>::MonopoleWithSymmetrySearch system;
```

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj>::QuadrupoleWithSymmetrySearch system;
```

各相互作用ツリークラスで使用している Moment クラスと SuperParticleJ クラスは、上記の順で、PS::MomentMonopoleSymmetry と PS::SPJMonopoleSymmetry (定義は 7.5.2.4.1, 7.6.2.3.1 参照)、PS::MomentQuadrupoleSymmetry と PS::SPJQuadrupoleSymmetry (定義は 7.5.2.4.2, 7.6.2.3.2 参照) である。

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.4 PS::SEARCH_MODE_LONG_CUTOFF

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj, TMom, TSpj>::WithCutoff system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラス、ローカルツリー及びグローバルツリーの Moment クラス、SuperParticleJ クラスである。

Moment クラスと SuperParticleJ クラスに FDPS が提供する PS::MomentMonopoleCutoff と PS::SPJMonopoleCutoff (定義は、それぞれ 7.5.2.5.1 節と 7.6.2.4.1 節) を使う場合には、以下の宣言で system を生成できる。これは、モーメント計算の中心を粒子の重心とした場合に、単極子まで計算するものである。

```
PS::TreeForForceLong
    <TResult, TEpi, TEpj>::MonopoleWithCutoff system;
```

テンプレート引数は PS::SEARCH_MODE_LONG の場合と同じである。

9.1.4.1.5 PS::SEARCH_MODE_GATHER

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Gather system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.6 PS::SEARCH_MODE_SCATTER

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Scatter system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.1.7 PS::SEARCH_MODE_SYMMETRY

以下のようにオブジェクト system を生成する。

```
PS::TreeForForceShort<TResult, TEpi, TEpj>::Symmetry system;
```

テンプレート引数は順に、Force クラス、EssentialParticleI クラス、EssentialParticleJ クラスである。

9.1.4.2 API

このモジュールには初期設定関連の API、相互作用計算関連の低レベル API、相互作用計算関連の高レベル API、ネイバースト関連の API がある。以下、各節に分けて記述する。

本節の中の API の宣言ではテンプレートクラスのテンプレート引数は省略する。すなわち、本来ならば以下のように記述するべきであるが、

```
template <class TSearchMode,
          class TResult,
          class TEpi,
          class TEpj,
          class TMomLocal,
          class TMomGlobal,
          class TSpj>
void PS::TreeForForce<TSearchMode,
                     TEpi,
                     TEpj,
                     TMomLocal,
                     TMomGlobal,
                     TSpj>::MemberFunction1();

template <class TSearchMode,
          class TResult,
          class TEpi,
          class TEpj,
          class TMomLocal,
          class TMomGlobal,
          class TSpj>
template <class TTT>
void PS::TreeForForce<TSearchMode,
                     TEpi,
                     TEpj,
                     TMomLocal,
                     TMomGlobal,
                     TSpj>::MemberFunction2(TTT arg1);
```

冗長であるので、以下のように省略する。

```
void PS::TreeForForce::MemberFunction1();

template <class TTT>
void PS::TreeForForce::MemberFunction2(TTT arg1);
```

9.1.4.2.1 初期設定

初期設定関連のAPIの宣言は以下のようになっている。このあと各APIについて記述する。

ソースコード 39: TreeForForce1

```
1 namespace ParticleSimulator {
2     template<class TSearchMode,
3             class TResult,
4             class TEpi,
5             class TEpj,
6             class TMomLocal,
7             class TMomGlobal,
8             class TSpj>
9     class TreeForForce{
10    public:
11        void TreeForForce();
12        void initialize(const U64 n_glb_tot,
13                      const F32 theta=0.7,
14                      const U32 n_leaf_limit=8,
15                      const U32 n_group_limit=64);
16        void setCommInfo(const CommInfo & comm);
17
18    };
19 }
```

9.1.4.2.1.1 コンストラクタ コンストラクタ

```
void PS::TreeForForce::TreeForForce();
```

- 引数
なし

- 返値
なし
- 機能
相互作用ツリークラスのオブジェクトを生成する。

9.1.4.2.1.2 *PS::TreeForForce::initialize*

PS::TreeForForce::initialize

```
void PS::TreeForForce::initialize
    (const PS::U64 n_glb_tot,
     const PS::F32 theta=0.7,
     const PS::U32 n_leaf_limit=8,
     const PS::U32 n_group_limit=64);
```

- 引数
n_glb_tot: 入力。const PS::U64 型。粒子配列の上限。
theta: 入力。const PS::F32 型。見こみ角に対する基準。デフォルト 0.7。
n_leaf_limit。const PS::U32 型。ツリーを切るのをやめる粒子数の上限。デフォルト 8。
n_group_limit。const PS::U32 型。相互作用リストを共有する粒子数の上限。デフォルト 64。
- 返値
なし
- 機能
相互作用ツリークラスのオブジェクトを初期化する。

9.1.4.2.1.3 *PS::TreeForForce::setCommInfo*

PS::TreeForForce::setCommInfo

```
void setCommInfo(const CommInfo & comm);
```

- 引数
comm: 入力。CommInfo &型。
- 返値
なし

- 機能

通信に使うコミュニケーターを指定する。

9.1.4.2.1.4 PS::TreeForForce::setExchangeLETMode

PS::TreeForForce::setExchangeLETMode

```
void setExchangeLETMode(enum PS::EXCHANGE_LET_MODE elm)
```

- 引数

elm: 入力 . enum PS::EXCHANGE_LET_MODE 型 .

- 返値

なし

- 機能

LET 交換の方法を elm で指定することができる . 許される入力は、6.8.4 で挙げた列挙型のみ。呼ばない場合は、PS::EXCHANGE_LET_A2A が使われる .

9.1.4.2.2 低レベル関数

相互作用計算関連の低レベル API の宣言は以下のようにになっている。このあと各 API について記述する。

ソースコード 40: TreeForForce1

```
1 namespace ParticleSimulator {
2     template<class TSearchMode,
3             class TResult,
4             class TEpi,
5             class TEpj,
6             class TMomLocal,
7             class TMomGlobal,
8             class TSpj>
9     class TreeForForce{
10     public:
11         template<class Tfunc_ep_ep>
12         void calcForce(Tfunc_ep_ep pfunc_ep_ep,
13                       const bool clear=true);
14         template<class Tfunc_ep_ep, class Tfunc_sp_ep>
15         void calcForce(Tfunc_ep_ep pfunc_ep_ep,
```



```

16             Tfunc_sp_ep pfunc_sp_ep,
17             const bool clear=true);
18     Tforce getForce(const S32 i);
19 };
20 }

```

9.1.4.2.2.1 PS::TreeForForce::calcForce

PS::TreeForForce::calcForce

```

template<class Tfunc_ep_ep>
void PS::TreeForForce::calcForce
    (Tfunc_ep_ep pfunc_ep_ep(TEpi *,
                             const PS::S32,
                             TEpj *,
                             const PS::S32,
                             TResult *),
     const bool clear=true);

```

- 引数

pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト (節 7.9 参照)。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TEpj *型、const PS::S32 型、TReturn *型。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- 返値

なし

- 機能

このオブジェクトに読み込まれた粒子すべての粒子間相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。

```

template<class Tfunc_ep_ep, class Tfunc_sp_ep>
void PS::TreeForForce::calcForce
    (Tfunc_ep_ep pfunc_ep_ep,
     Tfunc_ep_ep pfunc_sp_ep,
     const bool clear=true);

```

- 引数

pfunc_ep_ep: 入力。返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TEpj *型、const PS::S32 型、TResult *型。

pfunc_sp_ep: 入力。返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数オブジェクト。関数の引数は第 1 引数から順に (const) TEpi *型、const PS::S32 型、(const) TSpj *型、const PS::S32 型、TResult *型。

clear: 入力。const bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

- 返値

なし

- 機能

このオブジェクトに読み込まれた粒子すべての粒子間相互作用を計算する。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合に限る。

9.1.4.2.2.2 PS::TreeForForce::getForce

PS::TreeForForce::getForce

```
TResult PS::TreeForForce::getForce(const PS::S32 i);
```

- 引数

i: 入力。const PS::S32 型。粒子配列のインデックス。

- 返値

TResult 型。PS::TreeForForce::setParticleLocalTree で i 番目に読み込まれた粒子の受ける作用。

- 機能

PS::TreeForForce::setParticleLocalTree で i 番目に読み込まれた粒子の受ける作用を返す。

9.1.4.2.3 高レベル関数

相互作用計算関連の高レベル API の宣言は以下になっている。このあと各 API について記述する。

ソースコード 41: TreeForForce1

```
1 namespace ParticleSimulator {
2     template<class TSearchMode,
3             class TResult,
4             class TEpi,
5             class TEpj,
6             class TMomLocal,
7             class TMomGlobal,
8             class TSpj>
9     class TreeForForce{
10 public:
11
12     template <typename Tfunc_ep_ep, typename Head, typename...
13             Tails>
14     void calcForceAllAndWriteBack(Tfunc_ep_ep pfunc_ep_ep, Head
15             &&head, Tails &&...tails);
16
17     template <typename Tfunc_ep_ep, typename Head, typename...
18             Tails>
19     void calcForceAll(Tfunc_ep_ep pfunc_ep_ep, Head &&head,
20             Tails &&...tails);
21
22     template <typename Tfunc_dispatch, typename Tfunc_retrieve,
23             typename... Args>
24     S32 calcForceAllAndWriteBackMultiWalk(Tfunc_dispatch
25             pfunc_dispatch, Tfunc_retrieve pfunc_retrieve, const
26             S32 tag_max, Args &&...args);
27
28     template <typename Tfunc_dispatch, typename Tfunc_retrieve,
29             typename... Args>
30     S32 calcForceAllMultiWalk(Tfunc_dispatch pfunc_dispatch,
31             Tfunc_retrieve pfunc_retrieve, const S32 tag_max,
32             Args &&...args);
33
34     template <typename Tfunc_dispatch, typename Tfunc_retrieve,
35             typename... Args>
36     S32 calcForceAllAndWriteBackMultiWalkIndex(Tfunc_dispatch
37             pfunc_dispatch, Tfunc_retrieve pfunc_retrieve, const
38             S32 tag_max, Args &&...args);
39 }
```

```

27     template <typename Tfunc_dispatch, typename Tfunc_retrieve,
           typename... Args>
28     S32 calcForceAllMultiWalkIndex(Tfunc_dispatch
           pfunc_dispatch, Tfunc_retrieve pfunc_retrieve, const
           S32 tag_max, Args &&...args);
29
30     template <typename Tfunc_ep_ep, typename Head, typename...
           Tails>
31     void calcForceAndWriteBack(Tfunc_ep_ep pfunc_ep_ep, Head &&
           head, Tails &&...tails);
32
33 }
34 namespace PS = ParticleSimulator;

```

9.1.4.2.3.1 PS::TreeForForce::calcForceAllAndWriteBack

PS::TreeForForce::calcForceAllAndWriteBack

```

template <typename Tfunc_ep_ep, typename Head, typename... Tails>
void calcForceAllAndWriteBack(Tfunc_ep_ep pfunc_ep_ep, Head &&head,
                               Tails &&...tails);

```

• 引数

可変引数テンプレートを用いて任意個の引数を扱えるようになっており、引数の順序は以下ようになる。calcForceAllAndWriteBack(相互作用関数..., 粒子群クラスのオブジェクト..., DomainInfo dinfo, bool clear, PS::INTERACTION_LIST_MODE list_mode)。

先頭の引数は相互作用関数であり、第1引数は常に返値が void 型の EssentialParticleI と EssentialParticleJ の間の相互作用計算用の関数である。サーチモードが Long の場合は、2 番目の引数に返値が void 型の EssentialParticleI と SuperParticleJ の間の相互作用計算用の関数を指定する必要がある。相互作用関数の形は EssentialParticleI と EssentialParticleJ の場合は、

```
void calcForce(Tepi*, const PS::S32, Tepj*, const PS::S32, TResult*)
```

であり、EssentialParticleI と SuperParticleJ の場合は、

```
void calcForce(Tepi*, const PS::S32, TSuperParticleJ*, const PS::S32, TResult*)
```

である．

相互作用関数の次の引数は任意個の粒子群クラスのオブジェクトを指定できる．計算の結果はこれらのオブジェクトに書き戻される．これらの引数の後に，`dinfo`，`clear`，`list_mode` の意味は以下の通りである．また，`clear`，`list_mode` は省略が可能である．

`dinfo`: 入力．`DomainInfo` &型．領域クラスのオブジェクト．

`clear`: 入力．`bool` 型．前に計算された相互作用の結果をクリアするかどうかを決定するフラグ．`true` ならばクリアする．デフォルト `true` ．

`list_mode`: 入力．`PS::INTERACTION_LIST_MODE` 型．相互作用リストを作成し相互作用計算を行うか，前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ．`PS::MAKE_LIST` ならば新たに相互作用リストを作成する．この場合，次の相互作用計算時に相互作用リストの再利用はできず，新たに相互作用リストを作成する必要がある．`PS::MAKE_LIST_FOR_REUSE` は新たに相互作用リストを作成し相互作用計算を行う．この場合，次の相互作用計算時に，今回作った相互作用リストを再利用し相互作用計算ができる．`PS::REUSE_LIST` ならば，前回 `PS::MAKE_LIST_FOR_REUSE` を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う．デフォルトは `PS::MAKE_LIST` ．`PS::INTERACTION_LIST_MODE` の詳細はセクション 6.8.3 を参照．

- 返値

なし

- 機能

粒子群クラスのオブジェクト `psys` の粒子すべての相互作用を計算し，その計算結果を `psys` に書き戻す．

`PS::INTERACTION_LIST_MODE` により相互作用リストを新たに作るか再利用するかを選ぶことができる．相互作用リストの再利用している間は `PS::ParticleSystem::exchangeParticle()` によって粒子の交換を行ってはならない．また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない．

- 使用例

```

PS::ParticleSystem<FP0> psys0;
PS::ParticleSystem<FP1> psys1;
PS::ParticleSystem<FP2> psys2;
PS::ParticleSystem<FP3> psys3;
PS::ParticleSystem<FP4> psys4;
// 長距離力の場合 .
// 相互作用関数は 2 つ . その後に粒子群クラスを 2 つ (psys0 と psys1) 指定 .
tree_long.calcForceAllAndWriteBack(calcForceEpLong, calcForceSp, psys0,
psys1, dinfo, true, PS::MAKE_LIST);

// 単距離力の場合 .
// 相互作用関数は 1 つ . その後に粒子群クラスを 3 つ (psys2 , psys3, psys4) 指
定 .
tree_short.calcForceAllAndWriteBack(calcForceEpShort, psys2, psys3,
psys4, dinfo);

```

9.1.4.2.3.2 PS::TreeForForce::calcForceAll

PS::TreeForForce::calcForceAll

```

template <typename Tfunc_ep_ep, typename Head, typename... Tails>
void calcForceAll(Tfunc_ep_ep pfunc_ep_ep, Head &&head, Tails &&...tails);

```

- 引数

引数は関数 calcForceAllAndWriteBack と同じである .

- 返値

なし

- 機能

関数 calcForceAllAndWriteBack とほぼ同じであるが , 計算結果を粒子群クラスのオブジェクトに書き戻さない .

9.1.4.2.3.3 PS::TreeForForce::calcForceAllAndWriteBackMultiWalk

PS::TreeForForce::calcForceAllAndWriteBackMultiWalk

```
template <typename Tfunc_dispatch, typename Tfunc_retrieve, typename... Args>
S32 calcForceAllAndWriteBackMultiWalk(Tfunc_dispatch pfunc_dispatch,
Tfunc_retrieve pfunc_retrieve, const S32 tag_max, Args &&...args);
```

- 引数

可変引数テンプレートを用いて任意の引数を扱えるようになっており、引数の順序は以下
のようになる。calcForceAllAndWriteBack(Tfunc_dispatch pfunc_dispatch, Tfunc_retrieve
pfunc_retrieve, const S32 tag_max, 粒子群クラスのオブジェクト..., DomainInfo dinfo,
const PS::S32 n_walk_limit, bool clear, PS::INTERACTION_LIST_MODE list_mode)。

pfunc_dispatch: 入力。返値が void 型で、EssentialParticleI の配列と EssentialParticleJ
の配列 (と SuperParticleJ の配列) をアクセラレータに転送し、カーネルを発行させる
関数。PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG,

PS::SEARCH_MODE_LONG_CUTOFF の場合にこの関数は以下の形ををしている必要
がある。

```
PS::S32 pfunc_dispatch(const PS::S32 tag,
                      const PS::S32 nwalk,
                      const TEpi** iptcl,
                      const PS::S32* ni,
                      const TEpj** jptcl_ep,
                      const PS::S32* nj_ep,
                      const TSpj** jptcl_sp,
                      const PS::S32* nj_sp);
```

戻り値は PS::S32 型とし、正常に終了した場合は 0 を返す。

PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER,
PS::SEARCH_MODE_SYMMETRY の場合にこの関数は以下の形ををしている必要が
ある。

```
PS::S32 pfunc_dispatch(const PS::S32 tag,
                      const PS::S32 nwalk,
                      const TEpi** iptcl,
                      const PS::S32* ni,
                      const TEpj** jptcl_ep,
                      const PS::S32* nj_ep);
```

戻り値は PS::S32 型とし、正常に終了した場合は 0 を返す。

pfunc_retrieve: 入力・返値が void 型で、pfunc_dispatch で転送したデータの結果を回収する関数。この関数は以下の形をしている必要がある。

```
void pfunc_retrieve(const PS::S32 tag,
                   const PS::S32 nwalk,
                   const PS::S32* ni,
                   TResult** force);
```

引数 tag は pfunc_dispatch() と pfunc_retrieve() を関係させるもので、pfunc_dispatch で計算された結果は同じ tag の値を持つ pfunc_retrieve() で回収される。

tag_max: 入力・const PS::S32 型。発行される tag の数の最大値。扱う tag の番号は 0 から tag_max-1 までとなる。0 以下の整数を指定した場合はエラーを出力する。現バージョンでは、1 の場合に正常に動作し、1 を超える値を指定した場合には tag の値は 0 のみである。

tag_max の後に任意個の粒子群クラスのオブジェクトを指定できる。計算の結果はこれらのオブジェクトに書き戻される。

dinfo: 入力・DomainInfo &型。領域クラスのオブジェクト。

n_walk_limit: 入力・PS::S32 型。1 度にアクセラレータに送る相互作用リストの数の最大値。

clear: 入力・bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

list_mode: 入力・PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。

PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

- 返値
なし
- 機能

粒子群クラスのオブジェクト `psys` の粒子すべての相互作用を計算し、その計算結果を `psys` に書き戻す。相互作用リストを作る際にマルチウォーク法を用い、一度に複数の相互作用リストを作成する。一度に作成する相互作用リストの数の最大値は `n_walk_limit` である。

`PS::INTERACTION_LIST_MODE` により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は `PS::ParticleSystem::exchangeParticle()` によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

9.1.4.2.3.4 *PS::TreeForForce::calcForceAllMultiWalk*

`PS::TreeForForce::calcForceAllMultiWalk`

```
template <typename Tfunc_dispatch, typename Tfunc_retrieve, typename... Args>
S32 calcForceAllMultiWalk(Tfunc_dispatch pfunc_dispatch, Tfunc_retrieve pfunc_retrieve
```

- 引数

引数は関数 `calcForceAllAndWriteBackMultiWalk` と同じである。

- 返値

なし

- 機能

関数 `calcForceAllMultiWalk` とほぼ同じであるが、計算結果を粒子群クラスのオブジェクトに書き戻さない。

9.1.4.2.3.5 *PS::TreeForForce::calcForceAllAndWriteBackMultiWalkIndex*

`PS::TreeForForce::calcForceAllAndWriteBackMultiWalkIndex`

```
template <typename Tfunc_dispatch, typename Tfunc_retrieve, typename... Args>
S32 calcForceAllAndWriteBackMultiWalkIndex(Tfunc_dispatch pfunc_dispatch, Tfunc_re
```

- 引数

可変引数テンプレートを用いて任意個の引数を扱えるようになっており、引数の順序は以下になる。`.calcForceAllAndWriteBack(Tfunc_dispatch pfunc_dispatch, Tfunc_retrieve pfunc_retrieve, const S32 tag_max, 粒子群クラスのオブジェクト..., DomainInfo dinfo, PS::S32 n_walk_limit, bool clear, PS::INTERACTION_LIST_MODE list_mode)`。

pfunc_dispatch: 入力・返値が void 型で, EssentialParticleI の配列と EssentialParticleJ の配列 (と SuperParticleJ の配列) をアクセラレータに転送し, カーネルを発行させる関数. PS::SEARCH_MODE 型が PS::SEARCH_MODE_LONG, PS::SEARCH_MODE_LONG_CUTOFF の場合にこの関数は以下の形ををしている必要がある.

```
PS::S32 pfunc_dispatch(const PS::S32 tag,
                      const PS::S32 nwalk,
                      const TEpi** iptcl,
                      const PS::S32* ni,
                      const PS::S32** id_jptcl_ep,
                      const PS::S32* nj_ep,
                      const PS::S32** id_jptcl_sp,
                      const PS::S32* nj_sp,
                      const TEpj* jptcl_ep,
                      const PS::S32 n_send_ep,
                      const TSpj* jptcl_sp,
                      const PS::S32 n_send_sp,
                      const bool send_ptcl);
```

戻り値は PS::S32 型とし, 正常に終了した場合は 0 を返す.

PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合にこの関数は以下の形ををしている必要がある.

```
PS::S32 pfunc_dispatch(const PS::S32 tag,
                      const PS::S32 nwalk,
                      const TEpi** iptcl,
                      const PS::S32* ni,
                      const PS::S32** id_jptcl_ep,
                      const PS::S32* nj_ep,
                      const TEpj* jptcl_ep,
                      const PS::S32 n_send_ep,
                      const bool send_ptcl);
```

戻り値は PS::S32 型とし, 正常に終了した場合は 0 を返す.

pfunc_retrieve: 入力・返値が void 型で, pfunc_dispatch で転送したデータの結果を回収する関数. この関数は以下の形ををしている必要がある.

```
void pfunc_retrieve(const PS::S32 tag,
                   const PS::S32 nwalk,
                   const PS::S32* ni,
                   TResult** force);
```

引数 `tag` は `pfunc_dispatch()` と `pfunc_retrieve()` を関係させるもので、`pfunc_dispatch` で計算された結果は同じ `tag` の値を持つ `pfunc_retrieve()` で回収される。

`tag_max`: 入力、const PS::S32 型、発行される `tag` の数の最大値、扱う `tag` の番号は 0 から `tag_max-1` までとなる、0 以下の整数を指定した場合はエラーを出力する、現バージョンでは、1 の場合に正常に動作し、1 を超える値を指定した場合には `tag` の値は 0 のみである。

`tag_max` の後に任意個の粒子群クラスのオブジェクトを指定できる、計算の結果はこれらのオブジェクトに書き戻される。

`dinfo`: 入力、DomainInfo &型、領域クラスのオブジェクト。

`n_walk_limit`: 入力、const PS::S32 型、1 度にアクセラレータに送る相互作用リストの数の最大値。

`clear`: 入力、const bool 型、前に計算された相互作用の結果をクリアするかどうかを決定するフラグ、true ならばクリアする、デフォルト true。

`list_mode`: 入力、PS::INTERACTION_LIST_MODE 型、相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ、PS::MAKELIST ならば新たに相互作用リストを作成する、この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある、PS::MAKELIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う、この場合、次の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる、PS::REUSE_LIST ならば、前回 PS::MAKELIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う、デフォルトは PS::MAKELIST、PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

- 返値

なし

- 機能

粒子群クラスのオブジェクト `psys` の粒子すべての相互作用を計算し、その計算結果を `psys` に書き戻す。初期に全ての EPJ と SPJ を device メモリ上に転送し、FDPS はデバイス上のインデクスについての相互作用リストを作る。この際にマルチウォーク法を用い、一度に複数の相互作用リストを作成する。一度に作成する相互作用リストの数の最大値は `n_walk_limit` である。

PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

9.1.4.2.3.6 PS::TreeForForce::calcForceAllMultiWalkIndex

PS::TreeForForce::calcForceAllMultiWalkIndex

```
template <typename Tfunc_dispatch, typename Tfunc_retrieve, typename... Args>
S32 calcForceAllMultiWalkIndex(Tfunc_dispatch pfunc_dispatch, Tfunc_retrieve pfunc_retrieve, Args... args)
```

- 引数

引数は関数 `calcForceAllAndWriteBackMultiWalkIndex` と同じである。

- 返値

エラーコードを返す。現在は常に 0 を返す。

- 機能

関数 `calcForceAllMultiWalkIndex` とほぼ同じであるが、計算結果を粒子群クラスのオブジェクトに書き戻さない。

9.1.4.2.3.7 PS::TreeForForce::calcForceAndWriteBack

PS::TreeForForce::calcForceAndWriteBack

```
template <typename Tfunc_ep_ep, typename Head, typename... Tails>
void calcForceAndWriteBack(Tfunc_ep_ep pfunc_ep_ep, Head &&head, Tails &&...tails)
```

- 引数

可変引数テンプレートを用いて任意個の引数を扱えるようになっており、引数の順序は以下になる。`calcForceAndWriteBack(相互作用関数..., 粒子群クラスのオブジェクト..., bool clear, PS::INTERACTION_LIST_MODE list_mode)`。

先頭の引数は相互作用関数であり、第 1 引数は常に返値が `void` 型の `EssentialParticleI` と `EssentialParticleJ` の間の相互作用計算用の関数である。サーチモードが `Long` の場合は、2 番目の引数に返値が `void` 型の `EssentialParticleI` と `SuperParticleJ` の間の相互作用計算用の関数を指定する必要がある。相互作用関数の形は `EssentialParticleI` と `EssentialParticleJ` の場合は、

```
void calcForce(Tepi*, const PS::S32, Tepj*, const PS::S32, TResult*)
```

であり、`EssentialParticleI` と `SuperParticleJ` の場合は、

```
void calcForce(Tepi*, const PS::S32, TSuperParticleJ*, const PS::S32, TResult*)
```

である。

相互作用関数の次の引数は任意個の粒子群クラスのオブジェクトを指定できる。計算の結果はこれらのオブジェクトに書き戻される。これらの引数の後に、`dinfo`, `clear`, `list_mode` の意味は以下の通りである。また、`clear`, `list_mode` は省略が可能である。

`clear`: 入力、bool 型。前に計算された相互作用の結果をクリアするかどうかを決定するフラグ。true ならばクリアする。デフォルト true。

`list_mode`: 入力、PS::INTERACTION_LIST_MODE 型。相互作用リストを作成し相互作用計算を行うか、前回作成した相互作用リストを再利用し相互作用計算を行うかを決定するフラグ。PS::MAKE_LIST ならば新たに相互作用リストを作成する。この場合、次の相互作用計算時に相互作用リストの再利用はできず、新たに相互作用リストを作成する必要がある。PS::MAKE_LIST_FOR_REUSE は新たに相互作用リストを作成し相互作用計算を行う。この場合、次の相互作用計算時に、今回作った相互作用リストを再利用し相互作用計算ができる。PS::REUSE_LIST ならば、前回 PS::MAKE_LIST_FOR_REUSE を選んだ際に作成した相互作用リストを再利用し相互作用計算を行う。デフォルトは PS::MAKE_LIST。PS::INTERACTION_LIST_MODE の詳細はセクション 6.8.3 を参照。

- 返値

なし

- 機能

これより前に相互作用ツリークラスのオブジェクトに構築されたグローバルツリーとそのモーメントをもとに、相互作用ツリークラスのオブジェクトに属する粒子すべての相互作用が計算され、さらにその結果が粒子群クラスのオブジェクト `psys` に書き戻される。これを使うのは PS::SEARCH_MODE 型が PS::SEARCH_MODE_GATHER, PS::SEARCH_MODE_SCATTER, PS::SEARCH_MODE_SYMMETRY の場合に限る。PS::TreeForForce::calcForceAllAndWriteBack から粒子群クラスのオブジェクトの読み込み、ローカルツリーの構築、グローバルツリーの構築、グローバルツリーのモーメントの計算がなくなったもの。

PS::INTERACTION_LIST_MODE により相互作用リストを新たに作るか再利用するかを選ぶことができる。相互作用リストの再利用している間は PS::ParticleSystem::exchangeParticle() によって粒子の交換を行ってはならない。また、粒子データの消去や粒子配列の順番の並び替えを行ってはならない。

9.1.4.2.4 ネイバーリスト

9.1.4.2.4.1 getNeighborListOneParticle

getNeighborListOneParticle

```
template<class Tptcl>
PS::S32 PS::TreeForForce::getNeighborListOneParticle(const Tptcl & ptcl,
                                                    EPJ * & epj);
```

- 引数

ptcl: 入力。Tptcl &型。近傍粒子を求めたい粒子。

epj: 出力。EPJ * &型。近傍粒子の配列の先頭ポインタ。

- 返値

PS::S32 &型。近傍粒子の個数。

- 機能

呼び出し元のツリー構造を使って、ptcl の近傍粒子の配列の先頭ポインタを epj に与え、近傍粒子数を返す。epj は EPJ 型の粒子データの配列 (粒子へのポインタの配列ではない) の先頭へのポインタであり、FDPS が内部にもっているバッファ領域を指す。このため、ユーザはこのポインタに対して free() や delete() をしてはならない。この関数はスレッドセーフである。すなわち、スレッド毎に別のバッファ領域をもっている。なお、この領域はスレッド毎に 1 つしかないので、同一スレッドでこの関数を呼び出すたびに上書きされる。

これを使うのは PS::SEARCH_MODE 型が

- PS::SEARCH_MODE_GATHER
- PS::SEARCH_MODE_SCATTER
- PS::SEARCH_MODE_SYMMETRY
- PS::SEARCH_MODE_LONG_SCATTER
- PS::SEARCH_MODE_LONG_SYMMETRY
- PS::SEARCH_MODE_LONG_GATHER (未実装)
- PS::SEARCH_MODE_LONG_CUTOFF_GATHER (未実装)
- PS::SEARCH_MODE_LONG_CUTOFF_SCATTER (未実装)
- PS::SEARCH_MODE_LONG_CUTOFF_SYMMETRY (未実装)

の場合に限る。ptcl のメンバ関数には FP と同様に PS::F64vec getPos() が必要である。PS::SEARCH_MODE 型が

- PS::SEARCH_MODE_GATHER
- PS::SEARCH_MODE_SYMMETRY

- PS::SEARCH_MODE_LONG_SYMMETRY
- PS::SEARCH_MODE_LONG_GATHER (未実装)
- PS::SEARCH_MODE_LONG_CUTOFF_GATHER (未実装)
- PS::SEARCH_MODE_LONG_CUTOFF_SYMMETRY (未実装)

の場合にはさらに、ptcl の探査半径を返すメンバ関数 PS::F64 getRSearch() も必要となる。

9.1.4.2.5 時間計測

クラス内の時間計測関連の API の宣言は以下のようにになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバの time_profile_ の該当メンバに書き込む。メソッド clearTimeProfile() を呼ばない限り時間は足しあわされていく。

ソースコード 42: TreeForForce2

```

1 namespace ParticleSimulator {
2     template<class TSearchMode,
3             class TResult,
4             class TEpi,
5             class TEpj,
6             class TMomLocal,
7             class TMomGlobal,
8             class TSpj>
9     class TreeForForce{
10    public:
11        TimeProfile getTimeProfile();
12        void clearTimeProfile();
13    };
14 }
```

9.1.4.2.5.1 PS::TreeForForce::getTimeProfile

PS::TreeForForce::getTimeProfile

```
PS::TimeProfile PS::TreeForForce::getTimeProfile();
```

- 引数
なし。
- 返値

PS::TimeProfile 型。

- 機能

ローカルツリー構築、グローバルツリー構築、力の計算 (walk 込)、ローカルツリーのモーメント計算、グローバルツリーのモーメント計算、LET 構築、LET 交換にかかった時間 (ミリ秒単位) を TimeProfile 型のメンバ変数の該当部分 make_local_tree, make_global_tree_, calc_force_, calc_moment_local_tree_, calc_moment_global_tree_, make_LET_1st_, make_LET_2nd_, exchange_LET_1st_, exchange_LET_2nd_ に格納する。長距離力や散乱モードの様に LET 交換が 1 段階通信の場合は make_LET_2nd_, exchange_LET_2nd_ に値は格納されない。

9.1.4.2.5.2 PS::TreeForForce::clearTimeProfile

PS::TreeForForce::clearTimeProfile

```
void PS::TreeForForce::clearTimeProfile();
```

- 引数

なし。

- 返値

なし。

- 機能

相互作用ツリークラスの TimeProfile 型のプライベートメンバ変数のメンバ変数 make_local_tree, make_global_tree_, calc_force_, calc_moment_local_tree_, calc_moment_global_tree_, make_LET_1st_, make_LET_2nd_, exchange_LET_1st_, exchange_LET_2nd_ の値を 0 クリアする。

9.1.4.2.6 情報取得

クラス内の情報取得関連の API の宣言は以下のようにになっている。自クラスの主要なメソッドを呼び出すとそれにかかった時間をプライベートメンバの time_profile_ の該当メンバに書き込む。メソッド clearTimeProfile() を呼ばない限り時間は足しあわされていく。

ソースコード 43: TreeForForce2

```
1 namespace ParticleSimulator {  
2     template<class TSearchMode ,  
3         class TResult ,  
4         class TEpi ,  
5         class TEpj ,  
6         class TMomLocal ,
```



```

7         class TMomGlobal,
8         class TSpj>
9     class TreeForForce{
10    public:
11        TimeProfile getTimeProfile();
12        void clearTimeProfile();
13        Count_t getNumberOfInteractionEPEPLocal();
14        Count_t getNumberOfInteractionEPSPLocal();
15        Count_t getNumberOfInteractionEPEPGlobal();
16        Count_t getNumberOfInteractionEPSPGlobal();
17        void clearNumberOfInteraction();
18        S64 getUsedMemorySizeTotal();
19    };
20 }

```

9.1.4.2.6.1 *PS::TreeForForce::getNumberOfInteractionEPEPLocal*

PS::TreeForForce::getNumberOfInteractionEPEPLocal

PS::Count_t PS::TreeForForce::getNumberOfInteractionEPEPLocal();

- 引数
なし。
- 返値
PS::Count_t 型。
- 機能
自プロセス内で計算した EPI と EPJ の相互作用数を返す。

9.1.4.2.6.2 *PS::TreeForForce::getNumberOfInteractionEPEPGlobal*

PS::TreeForForce::getNumberOfInteractionEPEPGlobal

PS::Count_t PS::TreeForForce::getNumberOfInteractionEPEPGlobal();

- 引数
なし。

- 返値

PS::Count_t 型。

- 機能

全プロセス内で計算した EPI と EPJ の相互作用数を返す。

9.1.4.2.6.3 PS::TreeForForce::getNumberOfInteractionEPSPLocal

PS::TreeForForce::getNumberOfInteractionEPSPLocal

```
PS::Count_t PS::TreeForForce::getNumberOfInteractionEPSPLocal();
```

- 引数

なし。

- 返値

PS::Count_t 型。

- 機能

自プロセス内で計算した EPI と SPJ の相互作用数を返す。

9.1.4.2.6.4 PS::TreeForForce::getNumberOfInteractionEPSGlobal

PS::TreeForForce::getNumberOfInteractionEPSGlobal

```
PS::S64 PS::TreeForForce::getNumberOfInteractionEPSGlobal();
```

- 引数

なし。

- 返値

PS::Count_t 型。

- 機能

全プロセスで計算した EPI と SPJ の相互作用数を返す。

9.1.4.2.6.5 *PS::TreeForForce::clearNumberOfInteraction*

PS::TreeForForce::clearNumberOfInteraction

```
void PS::TreeForForce::clearNumberOfInteraction();
```

- 引数
なし。
- 返値
なし。
- 機能
EP-EP,EP-SP の local,global の相互作用数を 0 クリアする。

9.1.4.2.6.6 *PS::TreeForForce::getNumberOfWalkLocal*

PS::TreeForForce::getNumberOfWalkLocal

```
PS::Count\_t PS::TreeForForce::getNumberOfWalkLocal();
```

- 引数
なし。
- 返値
PS::Count_t 型。
- 機能
自プロセスでの相互作用計算時の tree walk 数を返す。

9.1.4.2.6.7 *PS::TreeForForce::getNumberOfWalkGlobal*

PS::TreeForForce::getNumberOfWalkGlobal

```
PS::Count\_t PS::TreeForForce::getNumberOfWalkGlobal();
```

- 引数
なし。
- 返値
PS::S64。

- 機能

全プロセスでの相互作用計算時の tree walk 数を返す。

9.1.4.2.6.8 PS::TreeForForce::getUsedMemorySize

PS::TreeForForce::getUsedMemorySize

```
PS::S64 PS::TreeForForce::getUsedMemorySize();
```

- 引数

なし。

- 返値

PS::S64。

- 機能

対象のオブジェクトが使用しているメモリー量を Byte 単位で返す。

9.1.4.2.7 粒子 id から EPJ を取得する

9.1.4.2.7.1 getEpjFromId

getEpjFromId

```
EPJ * PS::TreeForForce::getEpjFromId(const PS::S64 id)
```

- 引数

id: 入力。const PS::S64 型。取得したい粒子の id (EPJ のメンバ関数 getId() で返す値)。

- 返値

EPJ * 型: id に対応する EPJ の粒子ポインター。

- 機能

EPJ がメンバ関数 getId() を持つ場合に使用可能。引数 id と getId() で返した値が同じ EPJ のポインタを返す。対応する EPJ がない場合は NULL を返す。また、複数の EPJ が同じ id を持つ場合結果は保証されない。getId() についてはセクション 7.4.4.4.1 を参照。

9.1.4.2.8 Particle Mesh Multipole Method の API

9.1.4.2.8.1 setParamsPMMM

setParamsPMMM

```
void PS::TreeForForce::setParamsPMMM(const PS::S32vec & n_cell, PS::S32 i_cut)
```

- 引数

n_cell: 入力。const PS::S32 型。PMMM で使う規則格子の一次元方向の格子数。

i_cut: 入力。const PS::S32 型。隣接のメッシュ数。

- 返値

なし

- 機能

PMMM で使う規則格子の一次元方向の格子数と隣接のメッシュ数を設定する。ここで、n_cell は一次元方向の格子数を表し、例えば 8 に設定すると格子数は $8^3 = 512$ となる。また、i_cut は隣接のメッシュ数を表し、例えば 1 に設定すると隣接のメッシュは中心のメッシュを含めて $(1 \times 2 + 1)^3 = 27$ となる。PMMM の詳細はセクション?? を参照。

9.1.5 通信用データクラス

本節では、通信用データクラス CommInfo クラスと Comm クラスについて記述する。これらのクラスはノード間通信のための情報の保持や実際の通信を行うモジュールである。CommInfo クラスは 1 つの MPI のコミュニケータを持ち、そのメンバ関数はそのコミュニケータの元で動作する。ParticleSystem クラス、TreeForForce クラス、DomainInfo クラスはそれぞれ通信用データクラスをもっており、ユーザーは自分で作成した CommInfo クラスのオブジェクトをそれらのクラスのオブジェクトに与えることで、任意のコミュニケータ上で相互作用計算をさせることができる。

Comm クラスはコミュニケータに MPI_COMM_WORLD を指定した場合のラッパークラスとなっている。Comm クラスはシングルトンパターンで管理されており、ユーザーはオブジェクトの生成は必要としない。

ここでは CommInfo クラスと Comm クラスの API を記述する。

9.1.5.1 オブジェクトの生成

CommInfo クラスは以下のように宣言されている。

ソースコード 44: CommInfo0

```
1 namespace ParticleSimulator {
```

```
2     class CommInfo;
3 }
```

CommInfo クラスのオブジェクトの生成は以下のように行う。ここでは comm_info というオブジェクトを生成している。

```
PS::CommInfo comm_info;
```

9.1.5.2 API

CommInfo クラスの API の宣言は以下のようになっている。

ソースコード 45: CommunicationInformation

```
1 namespace ParticleSimulator {
2     class CommInfo{
3     public:
4         void setCommunicator(const MPI_Comm & comm);
5         CommInfo create(const int n, const int rank[]);
6         CommInfo split(int color, int key);
7         S32 getRank();
8         S32 getNumberOfProc();
9         S32 getRankMultiDim(const S32 id);
10        S32 getNumberOfProcMultiDim(const S32 id);
11        bool synchronizeConditionalBranchAND
12            (const bool local);
13        bool synchronizeConditionalBranchOR
14            (const bool local);
15        template<class T>
16        T getMinValue(const T val);
17        template<class Tfloat, class Tint>
18        void getMinValue(const Tfloat f_in,
19                        const Tint i_in,
20                        Tfloat & f_out,
21                        Tint & i_out);
22        template<class T>
23        T getMaxValue(const T val);
24        template<class Tfloat, class Tint>
25        void getMaxValue(const Tfloat f_in,
26                        const Tint i_in,
27                        Tfloat & f_out,
28                        Tint & i_out );
```

```

29     template<class T>
30     T getSum(const T val);
31     template<class T>
32     void broadcast(T * val,
33                   const S32 n,
34                   const S32 src=0);
35     MPI_Comm getCommunicator();
36 };
37 }

```

Comm クラスの API の宣言は以下のようになっている。

ソースコード 46: Communication

```

1 namespace ParticleSimulator {
2     class Comm{
3     public:
4         static S32 getRank();
5         static S32 getNumberOfProc();
6         static S32 getRankMultiDim(const S32 id);
7         static S32 getNumberOfProcMultiDim(const S32 id);
8         static bool synchronizeConditionalBranchAND
9             (const bool local);
10        static bool synchronizeConditionalBranchOR
11            (const bool local);
12        template<class T>
13        static T getMinValue(const T val);
14        template<class Tfloat, class Tint>
15        static void getMinValue(const Tfloat f_in,
16                                const Tint i_in,
17                                Tfloat & f_out,
18                                Tint & i_out);
19        template<class T>
20        static T getMaxValue(const T val);
21        template<class Tfloat, class Tint>
22        static void getMaxValue(const Tfloat f_in,
23                                const Tint i_in,
24                                Tfloat & f_out,
25                                Tint & i_out );
26        template<class T>
27        static T getSum(const T val);
28        template<class T>

```

```

29         static void broadcast(T * val,
30                                const S32 n,
31                                const S32 src=0);
32     };
33 }

```

9.1.5.2.1 初期設定

9.1.5.2.1.1 コンストラクタ コンストラクタ

```
PS::CommInfo PS::CommInfo::CommInfo(const MPI_Comm & comm = MPI_COMM_NULL);
```

- 引数

comm: 入力 . const MPI_Comm & 型 .

- 返値

CommInfo 型 . 生成されたコミュニケーター .

- 機能

CommInfo クラスのオブジェクトを生成する。デフォルトのコミュニケーターはMPI_COMM_NULL である .

9.1.5.2.2 PS::CommInfo::setCommunicator

```
void PS::CommInfo::setCommunicator(const MPI_Comm & comm = MPI_COMM_WORLD);
```

- 引数

comm: 入力 . const MPI_Comm & 型 .

- 返値

なし

- 機能

CommInfo クラスのオブジェクトに MPI コミュニケーターを設定する . デフォルトは MPI_COMM_WORLD である .

9.1.5.2.2.1 *PS::CommInfo::create*

PS::CommInfo::create

```
CommInfo create(const int n, const int rank[]) const;
```

- 引数

n: 入力．int 型．生成されるコミュニケータに所属するプロセスの数．

rank: 入力．int 型の配列．生成されるコミュニケータに所属するプロセスのランクの配列．

- 返値

CommInfo 型．生成されたコミュニケータ．

- 機能

呼び出しもとの CommInfo クラスのもつ MPI コミュニケータから新たなコミュニケータを作成する．配列 rank[] で表せられるプロセスが所属するコミュニケータを作成し返す．

9.1.5.2.2.2 *PS::CommInfo::split*

PS::CommInfo::split

```
CommInfo split(int color, int key) const;
```

- 引数

color: 入力．int 型．key: 入力．int 型．

- 返値

CommInfo 型．スプリットされたコミュニケータを返す．

- 機能

呼び出しもとの CommInfo クラスのもつ MPI コミュニケータを分割する．同じ color のプロセスは同一コミュニケータに所属し、key の小さいものから順にそのコミュニケータでのランクが割り振られる．

9.1.5.2.3 情報取得

9.1.5.2.3.1 *PS::CommInfo::getRank*

PS::CommInfo::getRank

```
PS::S32 PS::CommInfo::getRank();
```

- 引数
なし。
- 返回值
PS::S32 型。コミュニケータ内でのランクを返す。

9.1.5.2.3.2 PS::CommInfo::getNumberOfProc

PS::CommInfo::getNumberOfProc

```
PS::S32 PS::CommInfo::getNumberOfProc();
```

- 引数
なし。
- 返回值
PS::S32 型。全プロセス数を返す。

9.1.5.2.3.3 PS::CommInfo::getRankMultiDim

PS::CommInfo::getRankMultiDim

```
PS::S32 PS::CommInfo::getRankMultiDim(const PS::S32 id);
```

- 引数
id: 入力。const PS::S32 型。軸の番号。x 軸:0, y 軸:1, z 軸:2。
- 返回值
PS::S32 型。id 番目の軸でのランクを返す。2 次元の場合、id=2 は 1 を返す。

9.1.5.2.3.4 PS::CommInfo::getNumberOfProcMultiDim

PS::CommInfo::getNumberOfProcMultiDim

```
PS::S32 PS::CommInfo::getNumberOfProcMultiDim(const PS::S32 id);
```

- 引数

id: 入力。const PS::S32 型。軸の番号。x 軸:0, y 軸:1, z 軸:2。

- 返回值

PS::S32 型。id 番目の軸のプロセス数を返す。2 次元の場合、id=2 は 1 を返す。

9.1.5.2.3.5 PS::CommInfo::synchronizeConditionalBranchAND

PS::CommInfo::synchronizeConditionalBranchAND

```
bool PS::CommInfo::synchronizeConditionalBranchAND(const bool local)
```

- 引数

local: 入力。const bool 型。

- 返回值

bool 型。全プロセスで local の論理積を取り、結果を返す。

9.1.5.2.3.6 PS::CommInfo::synchronizeConditionalBranchOR

PS::CommInfo::synchronizeConditionalBranchOR

```
bool PS::CommInfo::synchronizeConditionalBranchOR(const bool local);
```

- 引数

local: 入力。const bool 型。

- 返回值

bool 型。全プロセスで local の論理和を取り、結果を返す。

9.1.5.2.3.7 PS::CommInfo::getMinValue

PS::CommInfo::getMinValue

```
template <class T>
T PS::CommInfo::getMinValue(const T val);
```

- 引数

val: 入力。const T 型。

- 戻り値

T 型。全プロセスで val の最小値を取り、結果を返す。

```
template <class Tfloat, class Tint>
void PS::CommInfo::getMinValue(const Tfloat f_in,
                               const Tint i_in,
                               Tfloat & f_out,
                               Tint & i_out);
```

- 引数

f_in: 入力。const Tfloat 型。

i_in: 入力。const Tint 型。

f_out: 出力。Tfloat 型。全プロセスで f_in の最小値を取り、結果を返す。

i_out: 出力。Tint 型。f_out に伴う ID を返す。

- 戻り値

なし。

9.1.5.2.3.8 PS::CommInfo::getMaxValue

PS::CommInfo::getMaxValue

```
template <class T>
static T PS::CommInfo::getMaxValue(const T val);
```

- 引数

val: 入力。const T 型。

- 戻り値

T 型。全プロセスで val の最大値を取り、結果を返す。

```
template <class Tfloat, class Tint>
void PS::CommInfo::getMaxValue(const Tfloat f_in,
                               const Tint i_in,
                               Tfloat & f_out,
                               Tint & i_out);
```

- 引数

f_in: 入力。const Tfloat 型。

i_in: 入力。const Tint 型。

f_out: 出力。Tfloat 型。全プロセスで f_in の最大値を取り、結果を返す。

i_out: 出力。Tint 型。f_out に伴う ID を返す。

- 返回值

なし。

9.1.5.2.3.9 PS::CommInfo::getSum

PS::CommInfo::getSum

```
template <class T>
T PS::Comm::getSum(const T val);
```

- 引数

val: 入力。const T 型。

- 返回值

T 型。全プロセスで val の総和を取り、結果を返す。

9.1.5.2.3.10 PS::CommInfo::broadcast

PS::CommInfo::broadcast

```
template <class T>
void PS::CommInfo::broadcast(T * val,
                             const PS::S32 n,
                             const PS::S32 src=0);
```

- 引数

val: 入力。T *型。

n: 入力。const PS::S32 型。T 型変数の数。

src: 入力。const PS::S32 型。放送するプロセスランク。デフォルトのランクは 0。

- 返回值

なし。

- 機能

プロセスランク `src` のプロセスが `n` 個の `T` 型変数を全プロセスに放送する。

9.1.5.2.3.11 *PS::Comm::getRank*

`PS::Comm::getRank`

```
static PS::S32 PS::Comm::getRank();
```

- 引数

なし。

- 返回值

`PS::S32` 型。全プロセス中でのランクを返す。

9.1.5.2.3.12 *PS::Comm::getNumberOfProc*

`PS::Comm::getNumberOfProc`

```
static PS::S32 PS::Comm::getNumberOfProc();
```

- 引数

なし。

- 返回值

`PS::S32` 型。全プロセス数を返す。

9.1.5.2.3.13 *PS::Comm::getRankMultiDim*

`PS::Comm::getRankMultiDim`

```
static PS::S32 PS::Comm::getRankMultiDim(const PS::S32 id);
```

- 引数

`id`: 入力。 `const PS::S32` 型。軸の番号。 `x` 軸:0, `y` 軸:1, `z` 軸:2。

- 返回值

`PS::S32` 型。 `id` 番目の軸でのランクを返す。2次元の場合、`id=2` は1を返す。

9.1.5.2.3.14 *PS::Comm::getNumberOfProcMultiDim*

PS::Comm::getNumberOfProcMultiDim

```
static PS::S32 PS::Comm::getNumberOfProcMultiDim(const PS::S32 id);
```

- 引数

id: 入力。const PS::S32 型。軸の番号。x 軸:0, y 軸:1, z 軸:2。

- 返回值

PS::S32 型。id 番目の軸のプロセス数を返す。2 次元の場合、id=2 は 1 を返す。

9.1.5.2.3.15 *PS::Comm::synchronizeConditionalBranchAND*

PS::Comm::synchronizeConditionalBranchAND

```
static bool PS::Comm::synchronizeConditionalBranchAND(const bool local)
```

- 引数

local: 入力。const bool 型。

- 返回值

bool 型。全プロセスで local の論理積を取り、結果を返す。

9.1.5.2.3.16 *PS::Comm::synchronizeConditionalBranchOR*

PS::Comm::synchronizeConditionalBranchOR

```
static bool PS::Comm::synchronizeConditionalBranchOR(const bool local);
```

- 引数

local: 入力。const bool 型。

- 返回值

bool 型。全プロセスで local の論理和を取り、結果を返す。

9.1.5.2.3.17 PS::Comm::getMinValue

PS::Comm::getMinValue

```
template <class T>
static T PS::Comm::getMinValue(const T val);
```

- 引数

val: 入力。const T 型。

- 返回值

T 型。全プロセスで val の最小値を取り、結果を返す。

```
template <class Tfloat, class Tint>
static void PS::Comm::getMinValue(const Tfloat f_in,
                                   const Tint i_in,
                                   Tfloat & f_out,
                                   Tint & i_out);
```

- 引数

f_in: 入力。const Tfloat 型。

i_in: 入力。const Tint 型。

f_out: 出力。Tfloat 型。全プロセスで f_in の最小値を取り、結果を返す。

i_out: 出力。Tint 型。f_out に伴う ID を返す。

- 返回值

なし。

9.1.5.2.3.18 PS::Comm::getMaxValue

PS::Comm::getMaxValue

```
template <class T>
static T PS::Comm::getMaxValue(const T val);
```

- 引数

val: 入力。const T 型。

- 返回值

T 型。全プロセスで val の最大値を取り、結果を返す。


```
template <class Tfloat, class Tint>
static void PS::Comm::getMaxValue(const Tfloat f_in,
                                   const Tint i_in,
                                   Tfloat & f_out,
                                   Tint & i_out);
```

- 引数

f_in: 入力。const Tfloat 型。

i_in: 入力。const Tint 型。

f_out: 出力。Tfloat 型。全プロセスで f_in の最大値を取り、結果を返す。

i_out: 出力。Tint 型。f_out に伴う ID を返す。

- 返回值

なし。

9.1.5.2.3.19 PS::Comm::getSum

PS::Comm::getSum

```
template <class T>
static T PS::Comm::getSum(const T val);
```

- 引数

val: 入力。const T 型。

- 返回值

T 型。全プロセスで val の総和を取り、結果を返す。

9.1.5.2.3.20 PS::Comm::broadcast

PS::Comm::broadcast

```
template <class T>
static void PS::Comm::broadcast(T * val,
                                 const PS::S32 n,
                                 const PS::S32 src=0);
```

- 引数

val: 入力。T *型。

n: 入力。const PS::S32 型。T 型変数の数。

src: 入力。const PS::S32 型。放送するプロセスランク。デフォルトのランクは 0。

- 返り値

なし。

- 機能

プロセスランク src のプロセスが n 個の T 型変数を全プロセスに放送する。

9.1.6 その他関数

本節では、名前空間 ParticleSimulator 以下に直に定義されている関数について述べる。

9.1.6.1 時間計測

9.1.6.1.1 PS::GetWtime

```
inline PS::F64 PS::GetWtime();
```

- 引数

なし。

- 返り値

PS::F64 型。ウォールクロックタイムを返す。単位は秒。

9.2 拡張機能

9.2.1 概要

本節では、FDPS の拡張機能について記述する。拡張機能には 2 つのモジュールがあり、Particle Mesh クラス、x86 版 Phantom-GRAPE がある。この 2 つのモジュールについて記述する。

9.2.2 Particle Mesh クラス

本節では、Particle Mesh クラスについて記述する。このクラスは Particle Mesh 法を用いて粒子の相互作用を計算するモジュールである。メッシュからの力は S-2 型の関数によってカットオフされており、カットオフ半径はメッシュ間隔の 3 倍で固定されている。カットオフ関数径を変更する事は出来ない。また、Particle Mesh クラスに送る粒子の座標は 0 以上 1

未満の値に規格化して置かなければならない。以下に、オブジェクトの生成方法、API、使用済マクロ、使いかたについて記述する。

9.2.2.1 オブジェクトの生成

Particle Mesh クラスは以下のように宣言されている。

ソースコード 47: ParticleMesh0

```
1 namespace ParticleSimulator {
2     namespace ParticleMesh {
3         class ParticleMesh;
4     }
5 }
```

Particle Mesh クラスのオブジェクトの生成は以下のように行う。ここでは pm というオブジェクトを生成している。

```
PS::PM::ParticleMesh pm;
```

9.2.2.2 API

Particle Mesh クラスには初期設定関連の API、低レベル API、高レベル API がある。以下、各節に分けて記述する。

9.2.2.2.1 初期設定

初期設定関連の API の宣言は以下のようになっている。このあと各 API について記述する。

ソースコード 48: ParticleMesh1

```
1 namespace ParticleSimulator {
2     namespace ParticleMesh {
3         class ParticleMesh{
4             ParticleMesh();
5         };
6     }
7 }
```

9.2.2.2.1.1 コンストラクタ

コンストラクタ

```
void PS::PM::ParticleMesh::ParticleMesh();
```

- 引数
なし
- 返値
なし
- 機能

Particle Mesh クラスのオブジェクトを生成する。

9.2.2.2.2 低レベル API

低レベル API の宣言は以下のようにになっている。このあと各 API について記述する。

ソースコード 49: ParticleMesh1

```
1 namespace ParticleSimulator {
2     namespace ParticleMesh {
3         class ParticleMesh{
4             template<class Tdinfo>
5             void setDomainInfoParticleMesh
6                 (const Tdinfo & dinfo);
7             template<class Tpsys>
8             void setParticleParticleMesh
9                 (const Tpsys & psys,
10                  const bool clear=true);
11             void calcMeshForceOnly();
12             F32vec getForce(F32vec pos);
13             F32 getPotential(F32vec pos);
14         };
15     }
16 }
```

9.2.2.2.2.1 *PS::PM::ParticleMesh::setDomainInfoParticleMesh*

PS::PM::ParticleMesh::setDomainInfoParticleMesh

```
template<class Tdinfo>
void PS::PM::ParticleMesh::setDomainInfoParticleMesh
    (const Tdinfo & dinfo);
```

- 引数
dinfo: 入力。Tdinfo &型。領域クラスのオブジェクト。
- 返値
なし
- 機能
領域情報を読み込む。

9.2.2.2.2.2 *PS::PM::ParticleMesh::setParticleParticleMesh*

PS::PM::ParticleMesh::setParticleParticleMesh

```
template<class Tpsys>
void PS::PM::ParticleMesh::setParticleParticleMesh
    (const Tpsys & psys,
     const bool clear=true);
```

- 引数
psys: 入力。Tpsys & 型。粒子群クラスのオブジェクト。
clear: 入力。const bool 型。これまで読込んだ粒子情報をクリアするかどうか決定するフラグ。true ならばクリアする。デフォルトは true。
- 返値
なし
- 機能
粒子情報を粒子群クラスのオブジェクトから読み込む。

9.2.2.2.2.3 *PS::PM::ParticleMesh::calcMeshForceOnly*

PS::PM::ParticleMesh::calcMeshForceOnly

```
void PS::PM::ParticleMesh::calcMeshForceOnly();
```

- 引数
なし
- 返値
なし
- 機能
メッシュ上の力を計算する。

9.2.2.2.2.4 *PS::PM::ParticleMesh::getForce*

PS::PM::ParticleMesh::getForce

```
PS::F32vec PS::PM::ParticleMesh::getForce(PS::F32vec pos);
```

- 引数
pos: 入力。PS::F32vec 型。メッシュからの力を計算したい位置。
- 返値
PS::F32vec 型。位置 pos におけるメッシュからの力。
- 機能
位置 pos でのメッシュからの力を返す。この関数は thread-safe である。

9.2.2.2.2.5 *PS::PM::ParticleMesh::getPotential*

PS::PM::ParticleMesh::getPotential

```
PS::F32 PS::PM::ParticleMesh::getPotential(PS::F32vec pos);
```

- 引数
pos: 入力。PS::F32vec 型。メッシュからのポテンシャルを計算したい位置。
- 返値
PS::F32 型。位置 pos におけるメッシュからポテンシャル。
- 機能
位置 pos でのメッシュからのポテンシャルを返す。この関数は thread-safe である。

9.2.2.2.3 高レベル API

高レベル API の宣言は以下のようにになっている。このあと各 API について記述する。

ソースコード 50: ParticleMesh1

```
1 namespace ParticleSimulator {  
2     namespace ParticleMesh {  
3         class ParticleMesh{  
4             template<class Tpsys ,  
5                 class Tdinfo>  
6                 void calcForceAllAndWriteBack  
7                     (Tpsys & psys ,  
8                     const Tdinfo & dinfo);  
9         };  
10    }  
11 }
```

9.2.2.2.3.1 *PS::PM::ParticleMesh::calcForceAllAndWriteBack*

PS::PM::ParticleMesh::calcForceAllAndWriteBack

```
template<class Tpsys ,  
        class Tdinfo>  
void PS::PM::ParticleMesh::calcForceAllAndWriteBack  
    (Tpsys & psys ,  
     const Tdinfo & dinfo);
```

- 引数

psys: 入力であり出力。Tpsys & 型。粒子群クラスのオブジェクト。

dinfo: 入力。const Tdinfo &型。領域クラスのオブジェクト。

- 返値

なし

- 機能

粒子群クラスのオブジェクト psys に含まれる粒子間のメッシュ力を計算し、その結果を psys に返す。

9.2.2.3 使用済マクロ

このモジュールでは多くのマクロを使っている。これらを別のマクロとして使用した場合にプログラムが正しく動作する保証はない。ここでは使用されているマクロをアルファベティカルに列挙する。

- BINARY_BOUNDARY
- BOUNDARY_COMM_NONBLOCKING
- BOUNDARY_SMOOTHING
- BUFFER_FOR_TREE
- CALCPOT
- CLEAN_BOUNDARY_PARTICLE
- CONSTANT_TIMESTEP
- EXCHANGE_COMM_NONBLOCKING
- FFT3D
- FFTW3_PARALLEL
- FFTW_DOUBLE
- FIX_FFTNODE
- GADGET_IO
- GRAPE_OFF
- KCOMPUTER
- LONG_ID
- MAKE_LIST_PROF
- MERGE_SNAPSHOT
- MULTI_TIMESTEP
- MY_MPLBARRIER
- N128_2H
- N256_2H
- N256_H

- N32_2H
- N512_2H
- NEW_DECOMPOSITION
- NOACC
- NPART_DIFFERENT_DUMP
- OMP_SCHEDULE_DISABLE
- PRINT_TANIKAWA
- REVERSE_ENDIAN_INPUT
- REVERSE_ENDIAN_OUTPUT
- RMM_PM
- SHIFT_INITIAL_BOUNDARY
- STATIC_ARRAY
- TREE2
- TREECONSTRUCTION_PARALLEL
- TREE_PARTICLE_CACHE
- UNIFORM
- UNSTABLE
- USING_MPI_PARTICLE
- VERBOSE_MODE
- VERBOSE_MODE2。

9.2.2.4 Particle Mesh クラスの使いかた

Particle Mesh クラスを使うには以下の4つのことを行う必要がある。

1. Particle Mesh クラスのコンパイル
2. Particle Mesh クラスを使った FDPS コードの記述
3. FDPS コードのコンパイル

以下、詳細に記述する。

9.2.2.4.1 Particle Mesh クラスのコンパイル

以下のように行う。ディレクトリ src の下のディレクトリ particle_mesh の Makefile を適切に編集して make する。編集すべきことは以下の2点である。

- INCLUDE_FFTW に FFTW のヘッダファイルがあるディレクトリを記述する
- param_fdps.h 中の SIZE_OF_MESH (1 次元方向のメッシュの数) を設定。推奨値は $N^{1/3}/2$ (N は粒子数)。

うまく行けば、同じディレクトリにライブラリ libpm.a とヘッダファイル particle_mesh.hpp ができる。

9.2.2.4.2 FDPS コードを記述

以下のように行う。

- 上でできたヘッダファイルを include する
- PM を計算したい粒子クラスに以下のメンバ関数を加える (この粒子クラスのクラス名を FP とする)
 - void FP::copyFromForceParticleMesh(const PS::F32vec & force)。この中で force を好きなメンバ変数にセットする。
 - PS::F64 FP::getChargeParticleMesh()。この中で質量を返す。
- このクラスのオブジェクトを生成するときに、PS::PM::ParticleMesh とする

9.2.2.4.3 FDPS コードのコンパイル

上で記述した FDPS コードをコンパイルするには以下のことを行う必要がある。

- ヘッダファイル particle_mesh.hpp のあるディレクトリへのパスを指定する
- ライブラリ libpm.a とリンクする
- FFTW のヘッダファイルがあるディレクトリへのパスを指定する
- FFTW のライブラリとリンクする

9.2.2.4.4 注意事項

Particle Mesh クラスはプロセス数が 2 以上でないと、動かないことに注意。

9.2.3 x86 版 phantom-GRAPE

低精度 N 体シミュレーション用、低精度カットオフ付き相互作用計算用の Phantom-GRAPE については Tanikawa et al.(2012, New Astronomy, 19, 74) を、高精度 N 体シミュレーション用の Phantom-GRAPE については Tanikawa et al.(2012, New Astronomy, 17, 82) を参照のこと。

9.2.4 Particle Mesh Multipole Moment クラス

本節では、ParticleMeshMultipole クラスについて記述する。このクラスは ParticleMeshMultipole 法を用いて粒子の相互作用を計算するモジュールである。ParticleMeshMultipole 法では、計算領域全体を規則格子に区切り、近傍セルからの寄与は直接計算し (PP パート)、遠方セルからの寄与は多重極展開を局所展開に変換して計算する (PM パート)。FDPS の標準モジュールでは、この PP パートをツリー法で計算する事ができ、外部モジュール ParticleMeshMultipole モジュールにより PM パートを計算する事ができる。ここでは、ParticleMeshMultipole モジュールについて記述する。ParticleMeshMultipole 法の詳細については、原論文 Nitadori(2014) を参照せよ。また、具体的な使用例については、FDPS のサンプルプログラムやチュートリアルを参照されたい。

以下に、オブジェクトの生成方法、API、使いかたについて記述する。

9.2.4.1 オブジェクトの生成

Particle Mesh Multipole クラスは以下のように宣言されている。

ソースコード 51: ParticleMeshMultipole0

```
1 namespace ParticleSimulator {  
2     namespace ParticleMeshMultipole {  
3         template <class Tforce, class Tepi>  
4             class ParticleMeshMultipole;  
5     }  
6 }
```

ParticleMeshMultipole クラスのオブジェクトの生成は以下のように行う。ここでは pmm というオブジェクトを生成している。テンプレート引数として Force 型と EPI 型を指定している。

```
PS::PMM::ParticleMeshMultipole pmm;
```

9.2.4.2 API

9.2.4.2.1 初期設定

初期設定関連の API の宣言は以下のようにになっている .

ソースコード 52: ParticleMesh1

```
1 namespace ParticleSimulator {  
2     namespace ParticleMeshMultipole {  
3         template <class Tforce, class Tepi>  
4         class ParticleMeshMultipole{  
5             void initialize(PS::S32 p_exp);  
6         };  
7     }  
8 }
```

9.2.4.2.1.1 コンストラクタ コンストラクタ

```
void PS::ParticleMesh::ParticleMeshMultipole();
```

- 引数
なし
- 返値
なし
- 機能

ParticleMeshMultipole クラスのオブジェクトを生成する。

9.2.4.2.1.2 *PS::PMM::ParticleMeshMultipole::initialize* PS::PMM::ParticleMeshMultipole::initialize

```
template<class Tdinfo>  
void PS::PMM::ParticleMeshMultipole::initialize(PS::S32 p_exp);
```

- 引数
p_exp: 入力 . PS::S32 型 . PMMM の展開次数 . 次数は任意 (0: 単極子, 1: 双極子, 2: 四重極子など).

- 返値

なし

- 機能

ParticleMeshMultipole クラスのオブジェクトの初期化する .

9.2.4.2.1.3 PS::PMM::ParticleMeshMultipole::calcForceAllAndWriteBack

PS::PMM::ParticleMeshMultipole::calcForceAllAndWriteBack

```
void calcForceAllAndWriteBack(TreeForForce<SEARCH_MODE_LONG_PARTICLE_MESH_MULTIPOLE, T> & tree,
                              const DomainInfo & dinfo,
                              Tpsys & psys,
                              const bool clear_force = true);
```

- 引数

tree: 入出力 . TreeForForce 型 . TreeForForce クラスのオブジェクト .

dinfo: 入力 . DomainInfo 型 . DomainInfo クラスのオブジェクト .

psys: 入出力 : ParticleSystemc 型 . 相互作用の結果を書き戻す粒子群のオブジェクト .

clear_force: 入力 . bool 型 . 粒子の力を初期化するかどうかを決定するフラグ . デフォルトは true .

- 返値

なし

- 機能

このクラスは ParticleMeshMultipole 法を用いて粒子の相互作用を計算する . 計算される値はポテンシャルとその勾配であり , Force::addForcePMMM を通して tree のメンバに値が加算された後 , その値が FullParticle::copyFromForcePMMM を通して psys に書き戻される .

9.2.4.3 ParticleMeshMultipole クラスの使いかた

ParticleMeshMultipole クラスはヘッダーライブラリであり , ヘッダファイル particle_mesh_multipole.h をインクルードすることで使用できる .

ParticleMeshMultipole クラスを使用した PM³ の計算の流れは以下である .

まず , FullParticle クラス , EssentialParticleI(J) クラス , Force クラス , 相互作用関数を定義する . PM³ を使用するためには , 標準モジュールが必要とするメンバ関数に加えて以下の関数が必要になる .

- `Force::addForcePMMM` — PM^3 モジュール内で, ツリークラスが持つ `Force` 型のメンバに PM^3 モジュールが計算した値を加算する.
- `FullParticle::copyFromForcePMMM` — PM^3 モジュール内で, `Force::addForcePMMM` が書き込んだ後の値を粒子群クラスにコピーする.

定義後, メインプログラムをで以下の手順で記述する.

粒子群クラスのオブジェクトを生成し, 粒子の位置, 質量, 速度などの初期値を設定する. 領域分割クラスのオブジェクトを作成・初期化し, `DomainInfo::setBoundaryCondition` を呼び出し周期境界条件を設定し (`PS::BOUNDARY_CONDITION_PERIODIC_XYZ`), `DomainInfo::setPosRoot` で領域サイズを指定する. その後, `DomainInfo::decomposeDomainAll`, `ParticleSystem::exchangePart` 等で領域分割や粒子交換を行う.

`TreeForForce` クラスのオブジェクトは `SearchMode` を `PS::SEARCH_MODE_LONG_PARTICLE_MESH` にし生成し初期化する. この際, モーメントクラスと `SPJ` クラスは `FDPS` が用意している以下のクラスを使用する必要がある.

- `MomentMonopole` と `SPJMonopole`
- `MomentQuadrupole` と `SPJQuadrupole`
- `MomentMonopoleGeometricCenter` と `SPJMonopoleGeometricCenter`
- `MomentDipoleGeometricCenter` と `SPJDipoleGeometricCenter`
- `MomentQuadrupoleGeometricCenter` と `SPJQuadrupoleGeometricCenter`

`TreeForForce` オブジェクトの作成後, `TreeForForce::setParamPMMM` を呼び出して, 計算領域の 1 次元方向の分割数および, PP パートで考慮する近傍セルの数を設定する. 後は通常通り, `TreeForForce::calcForceAllAndWriteBack()` などで PP パートを計算する.

PP パートの計算後, `PMM` クラスのオブジェクトを生成する. この時テンプレート引数として `TreeForForce` クラスで使った `Force` 型と `EPI` 型を指定する. さらに, `ParticleMeshMultipole::initia` を呼び出して初期化する. この際 PM^3 の多重局展開の最高次数および, `SPJ` が持つモーメントの最高次数を設定する. その後, `ParticleMeshMultipole::calcForceAllAndWriteBack` を呼び出して PM パートを計算する. PM パートの計算は必ず PP パートの計算後に行う必要がある事に注意すること. これは, PP パートの計算のために作られた木構造や多重極子などが PM パートで再利用されるためである.

10 エラー検出

10.1 概要

FDPS ではのコンパイル時もしくは実行時のエラー検出機能を備えている。ここでは、FDPS で検出可能なエラーとその場合の対処について記述する。ただし、ここに記述されていないエラーも起こる可能性がある。(その場合は開発者に報告していただけると助かります。)

10.2 コンパイル時のエラー

10.3 実行時のエラー

FDPS が実行時エラーを検出すると標準エラー出力に以下のような書式でメッセージを出し、PS::Abort(-1) によってプログラムを終了する。

```
PS_ERROR: ERROR MESSAGE  
function: FUNCTION NAME, line: LINE NUMBER, file: FILE NAME
```

- *ERROR MESSAGE*
エラーメッセージ
- *FUNCTION NAME*
エラーが起こった関数の名前
- *LINE NUMBER*
エラーが起こった行番号
- *FILE NAME*
エラーが起こったファイルの名前

以下、FDPS で用意されている実行時エラーメッセージを列挙していく。

10.3.1 PS_ERROR: can not open input file

ユーザーがFDPSのファイル入力関数を使っており、ユーザーが指定した入力ファイルがなかった場合に表示される。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

```
input file: “入力ファイル名”
```

10.3.2 PS_ERROR: can not open output file

ユーザーがFDPSのファイル出力関数を使っており、ユーザーが指定した出力ファイルがなかった場合に表示される。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

output file: “出力ファイル名”

10.3.3 PS_ERROR: Do not initialize the tree twice

同一のツリーオブジェクトに対して関数 `PS::TreeForForce::initialize(...)` を2度呼び出した場合に表示される。同一のツリーオブジェクトに対して `PS::TreeForForce::initialize(...)` の呼び出しを一回にする。

10.3.4 PS_ERROR: The opening criterion of the tree must be ≥ 0.0

長距離力モードでツリーのオープニングクライテリオンに負の値が入力された場合に表示される。関数 `PS::TreeForForce::initialize(...)` を使ってオープニングクライテリオンに0以上の値を指定する必要がある。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

theta_ = “入力されたオープニングクライテリオンの値”
SEARCH_MODE: “対象となるツリーのサーチモードの型名”
Force: “対象となるツリーのフォースの型名”
EPI: “対象となるツリーのEPIの型名”
EPJ: “対象となるツリーのEPJの型名”
SPJ: “対象となるツリーのSPJの型名”

10.3.5 PS_ERROR: The limit number of the particles in the leaf cell must be > 0

長距離力モードでツリーのリーフセルの最大粒子数に負の値が入力された場合に表示される。関数 `PS::TreeForForce::initialize(...)` を使ってリーフセルの最大粒子数に正の整数を指定する必要がある。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

n_leaf_limit_=” 入力されたリーフセルの最大粒子数”
SEARCH_MODE: “対象となるツリーのサーチモードの型名”
Force: “対象となるツリーのフォースの型名”
EPI: “対象となるツリーの EPI の型名”
EPJ: “対象となるツリーの EPJ の型名”
SPJ: “対象となるツリーの SPJ の型名”

10.3.6 PS_ERROR: The limit number of particles in ip groups msut be >= that in leaf cells

長距離力モードでツリーのリーフセルの最大粒子数が i 粒子グループの粒子の最大数より大きかった場合に表示される。関数 PS::TreeForForce::initialize(...) を使って i 粒子グループの最大粒子数をリーフセルの最大粒子数以上にすることがある。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

n_leaf_limit_=” 入力されたリーフセルの最大粒子数”
n_grp_limit_=” 入力された i 粒子グループの内の最大粒子数”
SEARCH_MODE: “対象となるツリーのサーチモードの型名”
Force: “対象となるツリーのフォースの型名”
EPI: “対象となるツリーの EPI の型名”
EPJ: “対象となるツリーの EPJ の型名”
SPJ: “対象となるツリーの SPJ の型名”

10.3.7 PS_ERROR: The number of particles of this process is beyond the FDPS limit number

FDPS では 1 プロセスあたりに扱える粒子数は $2G$ ($G=2^{30}$) であり、それ以上の粒子を確保しようとした場合に表示される。この場合、プロセス数を増やすなどして、1 プロセスあたりの粒子数を減らす必要がある。

10.3.8 PS_ERROR: The forces w/o cutoff can be evaluated only under the open boundary condition

開放境界以外の条件下でカットオフなし長距離力を設定した場合に表示される。カットオフなし長距離力の計算では必ず、開放境界条件を使う。無限遠までの粒子からの力を計算したい場合はカットオフあり長距離力の計算を FDPS で行い、カットオフ外からの力の計算は外部モジュールである Particle Mesh を使う事ができる。

10.3.9 PS_ERROR: A particle is out of root domain

ユーザーが `PS::DomainInfo::setRootDomain(...)` 関数を用いてルートドメインを設定しており、粒子がそのルートドメインからはみ出していた場合に表示される。周期境界条件の場合はユーザーは粒子をルートドメイン内に収まるように位置座標をシフトする必要がある。FDPS では粒子をルートドメイン内にシフトする関数

`PS::ParticleSystem::adjustPositionIntoRootDomain(...)` を用意しており、それを使うこともできる。

エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

position of the particle=" 粒子の座標"
position of the root domain=" ルートドメインの座標"

10.3.10 PS_ERROR: The smoothing factor of an exponential moving average is must between 0 and 1.

ユーザーが `PS::DomainInfo::initialize(...)` 関数を用いて平滑化係数に 0 未満もしくは 1 を超える値を設定した場合に表示される。エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

The smoothing factor of an exponential moving average=" 平滑化係数の値"

10.3.11 PS_ERROR: The coodinate of the root domain is inconsistent.

ユーザーが `PS::DomainInfo::setPosRootDomain(...)` 関数を用いてルートドメインを設定した時に、ユーザーが設定した小さい側の頂点の座標の任意の成分が大きい側の頂点の対応する座標の値よりも大きかった場合に表示される。エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

The coordinate of the low vertex of the rood domain=" 小さい側の頂点の座標"
The coordinate of the high vertex of the rood domain=" 大きい側の頂点の座標"

10.3.12 PS_ERROR: Vector invalid accesse

Vector 型の [] 演算子で定義されている範囲外の成分にアクセスを行った場合に表示される。エラーメッセージのあとに以下のメッセージも標準エラー出力に表示される。

Vector element=" 指定した成分" is not valid

11 よく知られているバグ

12 限界

- FDPS 独自の整数型を用いる場合、GCC コンパイラと K コンパイラでのみ正常に動作することが保証されている。

13 ユーザーサポート

FDPSを使用したコード開発に関する相談は以下のメールアドレス fdps-support@mail.jmlab.jp で受け付けています。以下のような場合は各項目毎の対応をお願いします。

13.1 コンパイルできない場合

ユーザーには以下の情報提供をお願いします。

- コンパイル環境
- コンパイル時に出力されるエラーメッセージ
- ソースコード (可能ならば)

13.2 コードがうまく動かない場合

ユーザーには以下の情報提供をお願いします。

- 実行環境
- 実行時に出力されるエラーメッセージ
- ソースコード (可能ならば)

13.3 その他

思い通りの性能がでない場合やその他の相談なども、上のメールアドレスにお知らせください。

14 ライセンス

MIT ライセンスに準ずる。標準機能のみ使用する場合は、Iwasawa et al. (PASJ, 68, 54)、Namekata et al. (PASJ, 70, 70)、Iwasawa et al. (PASJ, 72, 13) の引用をお願いします。

拡張機能の Particle Mesh クラスは GreeM コード (開発者: 石山智明、似鳥啓吾) (Ishiyama, Fukushige & Makino 2009, Publications of the Astronomical Society of Japan, 61, 1319; Ishiyama, Nitadori & Makino, 2012 SC'12 Proceedings of the International Conference on High Performance Computing, Networking Storage and Analysis, No. 5) のモジュールを使用している。GreeM コードは Yoshikawa & Fukushige (2005, Publications of the Astronomical Society of Japan, 57, 849) で書かれたコードをベースとしている。Particle Mesh クラスを使用している場合は、上記 3 つの文献の引用をお願いします。

拡張機能の Particle Mesh Multipole クラスを使用する場合は Nitadori (2014, arXiv:1409.5981) の引用をお願いします。

拡張機能のうち x86 版 Phantom-GRAPe を使用する場合は Tanikawa et al. (2012, New Astronomy, 17, 82) と Tanikawa et al. (2012, New Astronomy, 19, 74) の引用をお願いします。

Copyright (c) <2015-> <FDPS development team>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

15 変更履歴

- 2015.03.13
 - 様々なユーザー定義クラスのメンバ関数で `getRsearch` となっていたものを `getRSearch` と訂正 (節 7)
 - 異常終了させる関数 `PS::Abort` についての記述を追加 (節 8)
- 2015.03.17
 - バージョン 1.0 リリース
- 2015.03.18
 - Particle Mesh クラス関連のライセンス事項を修正。
- 2015.03.20
 - `PS::Comm::broadcast` の記述が抜けていたので追加。
- 2015.04.01
 - Particle Mesh クラスはプロセス数が 2 以上でないと動かないという注意事項を追加。
- 2015.10.07
 - PM 法についての記述を追加。セクション 9.2.2。
- 2015.12.01
 - Multi Walk についての記述を追加。セクション A.10、A.11、9.1.4.2.3.1、7.11、7.12。
- 2016.02.07
 - `getNeighborListOneParticle` についての記述を追加。セクション 9.1.4.2.4。
 - それに伴い、`PS::MomentMonopoleScatter`、`PS::MomentQuadrupoleScatter` の記述を追加。セクション A.4.2.2。
 - さらに、これらのラッパーである `MonopoleWithScatter` および `QuadrupoleWithScatter` の記述をセクション 9.1.4.1.1 に追加。
- 2016.09.13
 - 粒子の追加、削除の API をセクション 9.1.3.2.5 に追加。
 - Vector 型の範囲外アクセスについての記述をセクション 10.3.12 に追加。

- 2016.10.11
 - Vector 型の演算子” \cdot ”の実装変更。それによる注意事項を追加セクション 6.4.4 に追加。
- 2016.11.04
 - ファイル読み込み API をセクション 9.1.3.2.3.1,9.1.3.2.3.2,9.1.3.2.3.3,9.1.3.2.3.4 に追加
- 2017.07.11
 - 相互作用リストの再利用に関する記述をセクション 9.1.4.2.3 に追加。
- 2017.09.06
 - 粒子群クラスの粒子配列の並び替えに関する記述をセクション 9.1.3.2.7.3 に追加。
 - 粒子の id 番号から EPJ の情報を取り出す関数に関する記述をセクション 7.4.4.4.1,9.1.4.2.7 に追加。
- 2017.11.08
 - FDPS 4.0 リリース
- 2018.01.16
 - FDPS 4.0b リリース
 - * メモリ割付に関するバグを修正。
 - * TreeForForce クラス等にデストラクタを追加。
- 2018.01.23
 - 粒子交換時に粒子データをシリアルライズする方法をセクション 7.2.4.6, 7.4.4.5, 7.6.4.2, 9.1.3.2.4.1, 9.1.4.2.3 に追加。
- 2018.05.25
 - FDPS 4.0c リリース
 - * Particle Mesh 拡張の実装から MPI の C++ binding を削除。
- 2018.06.13
 - FDPS 4.1 リリース
 - * Tree 法で $\theta = 0$ の場合にも超粒子が発生しうる場合があるバグを修正。
 - * オルソトープ型の記述を第 6.5.4 節に追加。

- * Moment クラスのセクションに、場合によっては必要なメンバ関数についての記述を追加。
- 2018.06.20
 - FDPS 4.1a リリース
 - * 仕様書 (本書) の変更履歴のセクションの記載もれを修正。
- 2018.08.2
 - 新しいAPI `getBoundaryCondition` を追加
 - API `collectSampleParticle` の引数 `weight` の意味についての説明を追加
- 2018.12.7
 - FDPS 5.0a リリース
 - * Moment クラス 及び SuperParticleJ クラスの節 (第 7.5, 7.6 節) に、
`PS::SEARCH_MODE_LONG_SCATTER`
`PS::SEARCH_MODE_LONG_SYMMETRY`
 に対して用意されている既存のクラスの説明を追加。
 - * FullParticle クラス 及び ヘッダクラスの節 (第 7.2, 7.8 節) に、ファイル入出力 API を使う場合に必要となるメンバ関数 `readBinary` 及び `writeBinary` についての説明を追加。同様、これらの実装例を、それぞれ第 A.1, A.7 節に追加。
- 2018.12.9
 - FDPS 5.0b リリース
 - * 仕様書の API `getNeighborListOneParticle` の記述が不正確であった問題を修正 (ソースコードには変更ありません)。
- 2019.1.25
 - FDPS 5.0c リリース
 - * 周期境界条件で、かつ、特定の粒子分布が与えられた場合に FDPS 内部の関数 `LinkCell` でエラーが発生する問題を修正。
- 2019.3.1
 - FDPS 5.0d リリース
 - * 孤立境界条件及び全方向周期境界条件以外の境界条件で、ツリーのルートセルの設定が正しく行われない問題を修正。

- * FDPS バージョン 5.0 から 5.0c において、一部の SEARCH_MODE で、PS::TimeProfile のメンバ変数 make_LET_1st, make_LET_2nd, exchange_LET_1st, exchange_LET_2nd に値が設定されない問題を修正。時間測定の範囲の変更のため、個々の変数の値は FDPS バージョン 4.1 以前のものとは互換性がないが、これらの和については互換性がある。
 - * FDPS が内部で使用するメモリプールのサイズが不足していた場合、セグメンテーション違反によりコードが停止してしまう問題を修正。
 - * 本リリースから FDPS 内で C++11 の機能を使用している。**そのため、C++ コンパイラ、及び、CUDA を利用する場合には CUDA コンパイラに適切なオプション (GNU gcc であれば -std=c++11) をつける必要がある。**
- 2019.3.7
 - TreeForForceLong<,,>::MonopoleWithCutoff に関する記述を改善。
- 2019.9.6
 - FDPS 5.0f リリース
 - * コンパイル時にマクロ PARTICLE_SIMULATOR_TWO_DIMENSION を定義した場合、コンパイルエラーになる問題を修正。
- 2019.9.10
 - FDPS 5.0g リリース
 - * コンパイル時にマクロ PARTICLE_SIMULATOR_TWO_DIMENSION を定義した場合、実行時エラーになる問題を修正。
- 2020.8.16
 - FDPS 6.0 リリース
 - * PIKG を導入
- 2020.8.18
 - FDPS 6.0a リリース
 - * 付属する PIKG のバージョンを v0.1b に更新
- 2020.8.19
 - FDPS 6.0b リリース
 - * N 体シミュレーションサンプルコード (sample/*/nbody) の実装を PIKG で生成したカーネルを使った場合に性能が出るように改善
- 2020.8.28

- FDPS 6.0b1 リリース
 - * サンプルコードで使用可能な初期条件ファイルの配布先が変更になったため、チュートリアルを修正
- 2020.9.02
 - FDPS 6.0b2 リリース
 - * サンプルコードの TreeForForce クラスのオブジェクトを初期化する関数の引数を修正
 - * 上記に合わせて、チュートリアルも修正
- 2021.5.20
 - ツリーの最大深さ変更に関する記述をセクション 4.5 に追加
- 2021.5.20
 - 粒子ソートの変更方法に関する記述をセクション 4.6 に追加
- 2021.5.26
 - ツリーウォークでの粒子間の距離の計算方法を変更するための記述をセクション 6.8.5 に追加
- 2021.6.6
 - MPI コミュニケータを指定する CommInfo クラスの記述を 9.1.5 節に追加。
- 2025.09.03
 - FDPS8.0 リリース
 - * 外部モジュール ParticleMeshMultipole を追加し、PM³ 法による計算に対応。
 - * TreeForForce::calcForceAll 系の関数を多引数対応に拡張。
 - * 以下の関数を廃止
 - ・ TreeForForce::setParticleLocalTree
 - ・ TreeForForce::calcForceMakingTree
 - ・ TreeForForce::makeLocalTree
 - ・ TreeForForce::makeGlobalTree
 - ・ TreeForForce::calcMomentGlobalTree
 - * TimeProfile のメンバ変数を変更。関数 TimeProfile::dump を追加。
 - * TimeProfile に演算子"/"と"*"を追加。
 - * PS::S32, PS::S64, PS::U32, PS::U64 の型の定義を変更。

A ユーザー定義クラスの実装例

A.1 FullParticle クラス

A.1.1 概要

FullParticle クラスは粒子情報すべてを持つクラスであり、節 2.3 の手順 0 で、粒子群クラスに渡されるユーザー定義クラスの 1 つである。ユーザーはこのクラスに対して、どのようなメンバ変数、メンバ関数を定義してもかまわない。ただし、FDPS から FullParticle クラスの情報にアクセスするために、ユーザーはいくつかの決まった名前のメンバ関数を定義する必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

A.1.2 前提

この節の中では、以下のように、FullParticle クラスとして FP というクラスを一例とする。FP という名前は自由に変えることができる。

```
class FP;
```

A.1.3 必要なメンバ関数

A.1.3.1 概要

常に必要なメンバ関数は FP::getPos と FP::copyFromForce である。FP::getPos は FullParticle の位置情報を FDPS に読み込ませるための関数で、FP::copyFromForce は計算された相互作用の結果を FullParticle に書き戻す関数である。これらのメンバ関数の記述例と解説を以下に示す。

A.1.3.2 FP::getPos

```
class FP {  
public:  
    PS::F64vec getPos() const;  
};
```

- 引数
なし

- 返値

PS::F32vec 型または PS::F64vec 型。FP クラスのオブジェクトの位置情報を保持したメンバ変数。

- 機能

FP クラスのオブジェクトの位置情報を保持したメンバ変数を返す。

A.1.3.3 FP::copyFromForce

```
class Force {
public:
    PS::F64vec acc;
    PS::F64    pot;
};
class FP {
public:
    PS::F64vec acceleration;
    PS::F64    potential;
    void copyFromForce(const Force & force) {
        this->acceleration = force.acc;
        this->potential    = force.pot;
    }
};
```

- 前提

Force クラスは粒子の相互作用の計算結果を保持するクラス。

- 引数

force: 入力。const Force &型。粒子の相互作用の計算結果を保持。

- 返値

なし。

- 機能

粒子の相互作用の計算結果を FP クラスへ書き戻す。Force クラスのメンバ変数 acc, pot がそれぞれ FP クラスのメンバ変数 acceleration, potential に対応。

- 備考

Force クラスというクラス名とそのメンバ変数名は変更可能。FP のメンバ変数名は変更可能。メンバ関数 FP::copyFromForce の引数名は変更可能。

A.1.4 場合によっては必要なメンバ関数

A.1.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合、粒子群クラスの ファイル入出力 API を用いる場合、粒子群クラスの API である ParticleSystem::adjustPositionIntoRootDomain を用いる場合、拡張機能の Particle Mesh クラスを用いる場合について必要となるメンバ関数を記述する。

A.1.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

A.1.4.2.1 FP::getRSearch

```
class FP {  
public:  
    PS::F64 search_radius;  
    PS::F64 getRSearch() const {  
        return this->search_radius;  
    }  
};
```

- 前提

FP クラスのメンバ変数 `search_radius` はある 1 つの粒子の近傍粒子を探す半径の大きさ。この `search_radius` のデータ型は PS::F32 型または PS::F64 型。

- 引数

なし

- 返値

PS::F32 型または PS::F64 型。FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数。

- 機能

FP クラスのオブジェクトの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- 備考

FP クラスのメンバ変数 `search_radius` の変数名は変更可能。

A.1.4.3 粒子群クラスのファイル入出力 API を用いる場合

粒子群クラスのファイル入出力 API である `ParticleSystem::readParticleAscii`, `ParticleSystem::writeParticleAscii`, `ParticleSystem::readParticleBinary`, `ParticleSystem::writeParticleBinary` を使用するときそれぞれ `readAscii`, `writeAscii`, `readBinary`, `writeBinary` というメンバ関数が必要となる (`readAscii`, `writeAscii`, `readBinary`, `writeBinary` 以外の名前を使うことも可能。詳しくは節 9.1.3.2.3 を参照)。以下、`readAscii`, `writeAscii`, `readBinary`, `writeBinary` の実装例について記述する。

A.1.4.3.1 *FP::readAscii*

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void readAscii(FILE *fp) {
        fscanf(fp, "%d%lf%lf%lf%lf", &this->id, &this->mass,
               &this->pos[0], &this->pos[1], &this->pos[2]);
    }
};
```

- 前提

粒子データの入力ファイルの 1 列目には FP クラスのメンバ変数 `id` を表すデータが、2 列目にはメンバ変数 `mass` を表すデータが、3、4、5 列めにはメンバ変数 `pos` の第 1、2、3 要素が、それ以降には列がないとする。ファイルの形式はアスキー形式とする。3 次元直交座標系を選択したとする。

- 引数

`fp`: `FILE *`型。粒子データの入力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

粒子データの入力ファイルから FP クラスの `id`、`mass`、`pos` の情報を読み取る。

- 備考

なし。

A.1.4.3.2 FP::writeAscii

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void writeAscii(FILE *fp) {
        fscanf(fp, "%d %lf %lf %lf %lf", this->id, this->mass,
            this->pos[0], this->pos[1], this->pos[2]);
    }
};
```

- 前提

粒子データの出力ファイルの1列目にはFPクラスのメンバ変数 `id` を表すデータが、2列目にはメンバ変数 `mass` を表すデータが、3、4、5列めにはメンバ変数 `pos` の第1、2、3要素が、それ以降には列がないとする。ファイルの形式はアスキー形式とする。3次元直交座標系を選択したとする。

- 引数

`fp`: `FILE *`型。粒子データの出力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

粒子データの出力ファイルへFPクラスのメンバ変数 `id`、`mass`、`pos` の情報を書き出す。

- 備考

なし。

A.1.4.3.3 *FP::readBinary*

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void readBinary(FILE *fp) {
        fread(&this->id, sizeof(PS::S32), 1, fp);
        fread(&this->mass, sizeof(PS::F64), 1, fp);
        fread(&this->pos, sizeof(PS::F64vec), 1, fp);
    }
};
```

- 前提

粒子データの入力ファイルには (複数の)FP クラスのデータが連続して、すなわち、任意の 2 つの FP クラスのデータ間に余分な隙間なく、格納されているとする。FP クラス 1 個分のデータは、メンバ変数 `id`, `mass`, `pos` の第 1、2、3 要素がこの順に連続して並んでいるものとし、`fp` の現在位置はある FP クラスのデータの先頭を指しているとする。ファイルの形式はバイナリー形式とする。3 次元直交座標系を選択したとする。

- 引数

`fp`: `FILE *`型。粒子データの入力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

粒子データの入力ファイルから FP クラスの `id`、`mass`、`pos` の情報を読み取る。

- 備考

なし。

A.1.4.3.4 *FP::writeBinary*

```
class FP {
public:
    PS::S32 id;
    PS::F64 mass;
    PS::F64vec pos;
    void writeBinary(FILE *fp) {
        fwrite(&this->id, sizeof(PS::S32), 1, fp);
        fwrite(&this->mass, sizeof(PS::F64), 1, fp);
        fwrite(&this->pos, sizeof(PS::F64vec), 1, fp);
    }
};
```

- 前提

粒子データの出力ファイルには (複数の)FP クラスのデータが連続して、すなわち、任意の 2 つの FP クラスのデータ間に余分な隙間なく、格納されているとする。引数 `fp` が指す現在位置は最後に出力された FP クラスのデータの末尾を指しているとする。今、FP クラス 1 個分のデータを、メンバ変数 `id`, `mass`, `pos` の第 1、2、3 要素がこの順に連続して並ぶ形で出力したい。ファイルの形式はバイナリー形式とする。3 次元直交座標系を選択したとする。

- 引数

`fp`: `FILE *`型。粒子データの出力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

粒子データの出力ファイルへ FP クラスのメンバ変数 `id`, `mass`, `pos` の情報を書き出す。

- 備考

なし。

A.1.4.4 ParticleSystem::adjustPositionIntoRootDomain を用いる場合

A.1.4.4.1 FP::setPos

```
class FP {  
public:  
    PS::F64vec pos;  
    void setPos(const PS::F64vec pos_new) {  
        this->pos = pos_new;  
    }  
};
```

- 前提

FP クラスのメンバ変数 `pos` は 1 つの粒子の位置情報。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec`。

- 引数

`pos_new`: 入力。 `const PS::F32vec` または `const PS::F64vec` 型。FDPS 側で修正した粒子の位置情報。

- 返値

なし。

- 機能

FDPS が修正した粒子の位置情報を FP クラスのオブジェクトの位置情報に書き込む。

- 備考

FP クラスのメンバ変数 `pos` の変数名は変更可能。メンバ関数 `FP::setPos` の引数名 `pos_new` は変更可能。`pos` と `pos_new` のデータ型が異なる場合の動作は保証しない。

A.1.4.5 Particle Mesh クラスを用いる場合

Particle Mesh クラスを用いる場合には、メンバ関数 `FP::getChargeParticleMesh` と `FP::copyFromForceParticleMesh` を用意する必要がある。以下にそれぞれの規定を記述する。

A.1.4.5.1 *FP::getChargeParticleMesh*

```
class FP {
public:
    PS::F64 mass;
    PS::F64 getChargeParticleMesh() const {
        return this->mass;
    }
};
```

- 前提

FP クラスのメンバ変数 `mass` は 1 つの粒子の質量または電荷の情報を持つ変数。データ型は `PS::F32` または `PS::F64` 型。

- 引数

なし。

- 返値

`PS::F32` 型または `PS::F64` 型。 1 つの粒子の質量または電荷の変数を返す。

- 機能

1 つの粒子の質量または電荷を表すメンバ変数を返す。

- 備考

FP クラスのメンバ変数 `mass` の変数名は変更可能。

A.1.4.5.2 *FP::copyFromForceParticleMesh*

```
class FP {
public:
    PS::F64vec accelerationFromPM;
    void copyFromForceParticleMesh(const PS::F32vec & acc_pm) {
        this->accelerationFromPM = acc_pm;
    }
};
```

- 前提

FP クラスのメンバ変数 `accelerationFromPM_pm` は 1 つの粒子の Particle Mesh による力の情報を保持する変数。この `accelerationFromPM_pm` のデータ型は `PS::F32vec` または `PS::F64vec`。

- 引数

acc_pm: const PS::F32vec 型または const PS::F64vec 型。1つの粒子の Particle Mesh による力の計算結果。

- 返値

なし。

- 機能

1つの粒子の Particle Mesh による力の計算結果をこの粒子のメンバ変数に書き込む。

- 備考

FP クラスのメンバ変数 acc_pm の変数名は変更可能。メンバ関数 FP::copyFromForceParticleMesh の引数 acc_pm の引数名は変更可能。

A.2 EssentialParticleI クラス

A.2.1 概要

EssentialParticleI クラスは相互作用の計算に必要な i 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleI クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。そのため、EssentialParticleI クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

A.2.2 前提

この節の中では、EssentialParticleI クラスとして EPI というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPI, FP というクラス名は変更可能である。

EPI と FP の宣言は以下の通りである。

```
class FP;  
class EPI;
```

A.2.3 必要なメンバ関数

A.2.3.1 概要

常に必要なメンバ関数は EPI::getPos と EPI::copyfromFP である。EPI::getPos は EPI クラスの位置情報を FDPS に読み込ませるための関数で、EPI::copyFromFP は FP クラスの情報を EPI クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

A.2.3.2 EPI::getPos

```
class EPI {  
public:  
    PS::F64vec pos;  
    PS::F64vec getPos() const {  
        return this->pos;  
    }  
};
```

- 前提

EPIのメンバ変数 `pos` はある1つの粒子の位置情報。この `pos` のデータ型は `PS::F64vec` 型。

- 引数

なし

- 返値

`PS::F64vec` 型。EPIクラスの位置情報を保持したメンバ変数。

- 機能

EPIクラスのオブジェクトの位置情報を保持したメンバ変数を返す。

- 備考

EPIクラスのメンバ変数 `pos` の変数名は変更可能。

A.2.3.3 EPI::copyFromFP

```
class FP {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};

class EPI {
public:
    PS::S64    id;
    PS::F64vec pos;
    void copyFromFP(const FP & fp) {
        this->id  = fp.identity;
        this->pos = fp.position;
    }
};
```

- 前提

FP クラスのメンバ変数 `identity`, `position` と EPI クラスのメンバ変数 `id`, `pos` はそれぞれ対応する情報を持つ。

- 引数

`fp`: 入力。 `const FP &`型。FP クラスの情報を持つ。

- 返値

なし。

- 機能

FP クラスの持つ 1 粒子の情報の一部を `EssnetialParticleI` クラスに書き込む。

- 備考

FP クラスのメンバ変数の変数名、EPI クラスのメンバ変数の変数名は変更可能。メンバ関数 `EPI::copyFromFP` の引数名は変更可能。EPI クラスの粒子情報は FP クラスの粒子情報のサブセット。対応する情報を持つメンバ変数同士のデータ型が一致している必要はないが、実数型とベクトル型 (または整数型とベクトル型) という違いがある場合に正しく動作する保証はない。

A.2.4 場合によっては必要なメンバ関数

A.2.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合に必要となるメンバ関数について記述する。

A.2.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_GATHER または PS::SEARCH_MODE_SYMMETRY を用いる場合

A.2.4.2.1 EPI::getRSearch

```
class EPI {  
public:  
    PS::F64 search_radius;  
    PS::F64 getRSearch() const {  
        return this->search_radius;  
    }  
};
```

- 前提

EPI クラスのメンバ変数 `search_radius` はある 1 つの粒子の近傍粒子を探す半径の大きさ。この `search_radius` のデータ型は PS::F32 型または PS::F64 型。

- 引数

なし

- 返値

PS::F32 型または PS::F64 型。 EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。

- 機能

EPI クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。

- 備考

EPI クラスのメンバ変数 `search_radius` の変数名は変更可能。

A.3 EssentialParticleJ クラス

A.3.1 概要

EssentialParticleJ クラスは相互作用の計算に必要な j 粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。EssentialParticleJ クラスは FullParticle クラス (節 7.2) のサブセットである。FDPS は、このクラスのデータにアクセスする必要がある。このために、EssentialParticleJ クラスはいくつかのメンバ関数を持つ必要がある。以下、この節の前提、常に必要なメンバ関数と、場合によっては必要なメンバ関数について記述する。

A.3.2 前提

この節の中では、EssentialParticleJ クラスとして EPJ というクラスを一例として使う。また、FullParticle クラスの一例として FP というクラスを使う。EPJ, FP というクラス名は変更可能である。

EPJ と FP の宣言は以下の通りである。

```
class FP;  
class EPJ;
```

A.3.3 必要なメンバ関数

A.3.3.1 概要

常に必要なメンバ関数は EPJ::getPos と EPJ::copyfromFP である。EPJ::getPos は EPJ クラスの位置情報を FDPS に読み込ませるための関数で、EPJ::copyFromFP は FP クラスの情報を EPJ クラスに書きこむ関数である。これらのメンバ関数の記述例と解説を以下に示す。

A.3.3.2 EPJ::getPos

```
class EPJ {  
public:  
    PS::F64vec pos;  
    PS::F64vec getPos() const {  
        return this->pos;  
    }  
};
```

- 前提

EPJ のメンバ変数 pos はある 1 つの粒子の位置情報。この pos のデータ型は PS::F64vec 型。

- 引数
なし
- 返値
PS::F64vec 型。EPJ クラスの位置情報を保持したメンバ変数。
- 機能
EPJ クラスの位置情報を保持したメンバ変数を返す。
- 備考
EPJ クラスのメンバ変数 `pos` の変数名は変更可能。

A.3.3.3 EPJ::copyFromFP

```
class FP {
public:
    PS::S64    identity;
    PS::F64    mass;
    PS::F64vec position;
    PS::F64vec velocity;
    PS::F64vec acceleration;
    PS::F64    potential;
};

class EPJ {
public:
    PS::S64    id;
    PS::F64    m;
    PS::F64vec pos;
    void copyFromFP(const FP & fp) {
        this->id  = fp.identity;
        this->m   = fp.mass;
        this->pos = fp.position;
    }
};
```

- 前提
FP クラスのメンバ変数 `identity`, `mass`, `position` と EPJ クラスのメンバ変数 `id`, `m`, `pos` はそれぞれ対応する情報を持つ。

- 引数

fp: 入力。const FP &型。FP クラスの情報を持つ。

- 返値

なし。

- 機能

FP クラスの持つ 1 粒子の情報の一部を EPJ クラスに書き込む。

- 備考

FP クラスのメンバ変数の変数名、EPJ クラスのメンバ変数の変数名は変更可能。メンバ関数 EPJ::copyFromFP の引数名は変更可能。対応する情報を持つメンバ変数同士のデータ型が一致している必要はないが、実数型とベクトル型 (または整数型とベクトル型) という違いがある場合に正しく動作する保証はない。

A.3.4 場合によっては必要なメンバ関数

A.3.4.1 概要

本節では、場合によっては必要なメンバ関数について記述する。相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合に必要なメンバ関数、列挙型の BOUNDARY_CONDITION 型に PS::BOUNDARY_CONDITION_OPEN 以外を選んだ場合に必要となるメンバ関数について記述する。なお、既存の Moment クラスや SuperParticleJ クラスを用いる際に必要となるメンバ変数はこれら既存のクラスの節を参照のこと。

A.3.4.2 相互作用ツリークラスの PS::SEARCH_MODE 型に PS::SEARCH_MODE_LONG 以外を用いる場合

A.3.4.2.1 EPJ::getRSearch

```
class EPJ {
public:
    PS::F64 search_radius;
    PS::F64 getRSearch() const {
        return this->search_radius;
    }
};
```

- 前提

EPJ クラスのメンバ変数 `search_radius` はある 1 つの粒子の近傍粒子を探す半径の大きさ。この `search_radius` のデータ型は `PS::F32` 型または `PS::F64` 型。

- 引数
なし
- 返値
`PS::F32` 型または `PS::F64` 型。EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数。
- 機能
EPJ クラスの近傍粒子を探す半径の大きさを保持したメンバ変数を返す。
- 備考
EPJ クラスのメンバ変数 `search_radius` の変数名は変更可能。

A.3.4.3 BOUNDARY_CONDITION 型に `PS::BOUNDARY_CONDITION_OPEN` 以外を用いる場合

A.3.4.3.1 *EPJ::setPos*

```
class EPJ {  
public:  
    PS::F64vec pos;  
    void setPos(const PS::F64vec pos_new) {  
        this->pos = pos_new;  
    }  
};
```

- 前提
EPJ クラスのメンバ変数 `pos` は 1 つの粒子の位置情報。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec`。EPJ クラスのメンバ変数 `pos` の元データとなっているのは FP クラスのメンバ変数 `position`。このデータ型は `PS::F32vec` または `PS::F64vec`。
- 引数
`pos_new`: 入力。 `const PS::F32vec` または `const PS::F64vec` 型。FDPS 側で修正した粒子の位置情報。
- 返値
なし。

- 機能

FDPS が修正した粒子の位置情報を EPJ クラスの位置情報に書き込む。

- 備考

EPJ クラスのメンバ変数 `pos` の変数名は変更可能。メンバ関数 `EPJ::setPos` の引数名 `pos_new` は変更可能。 `pos` と `pos_new` のデータ型が異なる場合の動作は保証しない。

A.4 Moment クラス

A.4.1 概要

Moment クラスは近い粒子同士でまとめた複数の粒子のモーメント情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。モーメント情報の例としては、複数粒子の単極子や双極子、さらにこれら粒子の持つ最大の大きさなど様々なものが考えられる。このクラスは、`EssentialParticleJ` クラスから `SuperParticleJ` クラスを作るための中間変数のような役割を果たす。従って、このクラスが持つメンバ関数は、`EssentialParticleJ` クラスから情報を読み出してモーメントを計算するメンバ関数、少ない数の粒子のモーメントからそれらの粒子を含むより多くの粒子のモーメントを計算するメンバ関数などがある。

このようなモーメント情報にはある程度決っているものが多いので、それらについては FDPS 側で用意した。これら既存のクラスについてまず記述する。その後ユーザーがモーメントクラスを自作する際に必ず必要なメンバ関数、場合によっては必要になるメンバ関数について記述する。

A.4.2 既存のクラス

A.4.2.1 概要

FDPS はいくつかの Moment クラスを用意している。これらは相互作用ツリークラスで特定の `PS::SEARCH_MODE` 型を選んだ場合に有効である。以下、各 `PS::SEARCH_MODE` 型において選ぶことのできる Moment 型を記述する。`PS::SEARCH_MODE_GATHER`, `PS::SEARCH_MODE_SCATTER`, `PS::SEARCH_MODE_SYMMETRY` については Moment クラスを意識してコーディングする必要がないので、これらについては記述しない。

A.4.2.2 `PS::SEARCH_MODE_LONG`

以下に述べる Moment クラスのうち、`MomentMonopole` 及び `MomentQuadrupole` については、近接粒子探索を行うことができる `PS::SEARCH_MODE_LONG_SCATTER` で使うものも同様に定義されている。それぞれ `MomentMonopoleScatter` および `MomentQuadrupoleScatter` である。

A.4.2.2.1 *PS::MomentMonopole*

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class MomentMonopole {  
    public:  
        F32    mass;  
        F32vec pos;  
    };  
}
```

- クラス名 PS::MomentMonopole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

MomentMonopoleScatter の場合、EssentialParticleJ クラスはメンバ関数 getRSearch() をもつこと。

A.4.2.2.2 *PS::MomentQuadrupole*

単極子と四重極子を情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class MomentQuadrupole {  
    public:  
        F32    mass;  
        F32vec pos;  
        F32mat quad;  
    };  
}
```

- クラス名 PS::MomentQuadrupole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量

pos: 近傍でまとめた粒子の重心

quad: 近傍でまとめた粒子の四重極子

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

MomentQuadrupoleScatter の場合、EssentialParticleJ クラスはメンバ関数 getRSearch() をもつこと。

A.4.2.2.3 PS::MomentMonopoleGeometricCenter

単極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class MomentMonopoleGeometricCenter {  
    public:  
        F32      charge;  
        F32vec pos;  
    };  
}
```

- クラス名 PS::MomentMonopoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

A.4.2.2.4 PS::MomentDipoleGeometricCenter

双極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```

namespace ParticleSimulator {
    class MomentDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}

```

- クラス名 PS::MomentDipoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

A.4.2.2.5 PS::MomentQuadrupoleGeometricCenter

四重極子までを情報として持つクラス。これらのモーメントを計算する際の座標系の中心には粒子の幾何中心を取る。以下、このクラスの概要を記述する。

```

namespace ParticleSimulator {
    class MomentQuadrupoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}

```

- クラス名 PS::MomentQuadrupoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

quadrupole: 粒子の質量または電荷の四重極子

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge と EssentialParticleJ::getPos を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

A.4.2.3 PS::SEARCH_MODE_LONG_CUTOFF

以下に述べる Moment クラスのうち、MomentMonopoleCutoff については、近接粒子探索を行うことができる PS::SEARCH_MODE_LONG_CUTOFF_SCATTER で使うものも同様に定義されている。MomentMonopoleCutoffScatter である。

A.4.2.3.1 PS::MomentMonopoleCutoff

単極子までを情報として持つクラス。単極子を計算する際の座標系の中心には粒子の重心や粒子電荷の重心を取る。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {  
    class MomentMonopoleCutoff {  
    public:  
        F32    mass;  
        F32vec pos;  
    };  
}
```

- クラス名 PS::MomentMonopoleCutoff

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

EssentialParticleJ クラス (節 7.4) がメンバ関数 EssentialParticleJ::getCharge, EssentialParticleJ::getPos, EssentialParticleJ::getRSearch を持ち、それぞれが粒子質量 (または粒子電荷)、粒子位置、粒子の力の到達距離を返すこと。EssentialParticleJ クラスのクラス名は変更自由。

MomentMonopoleCutoffScatter の場合、EssentialParticleJ クラスはメンバ関数 getRSearch() をもつこと。

A.4.3 必要なメンバ関数

A.4.3.1 概要

以下では Moment クラスを定義する際に、必要なメンバ関数を記述する。このとき Moment クラスのクラス名を Mom とする。これは変更自由である。

A.4.3.2 コンストラクタ

```
class Mom {  
public:  
    PS::F32    mass;  
    PS::F32vec pos;  
    Mom () {  
        mass = 0.0;  
        pos  = 0.0;  
    }  
};
```

- 前提

Mom クラスのメンバ変数 mass, pos は Mom の質量と位置。

- 引数

なし

- 返値

なし

- 機能

Mom クラスのオブジェクトの初期化をする。

- 備考

メンバ変数名の変更可能。メンバ変数を加えることも可能。

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom(const PS::F32 m,
         const PS::F32vec & p) {
        mass = m;
        pos  = p;
    }
};
```

- 前提

Mom クラスのメンバ変数 mass, pos は Mom の質量と位置。

- 引数

m: 入力。const PS::F32 型。質量

p: 入力。const PS::F32vec &型。位置。

- 返値

なし

- 機能

Mom クラスのオブジェクトの初期化をする。

- 備考

メンバ変数名の変更可能。メンバ変数を加えることも可能。

A.4.3.3 Mom::init

```
class Mom {
public:
    void init();
};
```

- 前提

なし

- 引数

なし

- 返値
なし
- 機能
Mom クラスのオブジェクトの初期化をする。
- 備考
なし

A.4.3.4 Mom::getPos

```
class Mom {
public:
    PS::F32vec pos;
    PS::F32vec getPos() const {
        return pos;
    }
};
```

- 前提
Mom のメンバ変数 pos は近傍でまとめた粒子の代表位置。この pos のデータ型は PS::F32vec または PS::F64vec 型。
- 引数
なし
- 返値
PS::F32vec または PS::F64vec 型。Mom クラスのメンバ変数 pos。
- 機能
Mom クラスのメンバ変数 pos を返す。
- 備考
Mom クラスのメンバ変数 pos の変数名は変更自由。

A.4.3.5 Mom::getCharge

```
class Mom {  
public:  
    PS::F32 mass;  
    PS::F32 getCharge() const {  
        return mass;  
    }  
};
```

- 前提

Mom のメンバ変数 `mass` は近傍でまとめた粒子の全質量または全電荷。この `mass` のデータ型は `PS::F32` または `PS::F64` 型。

- 引数

なし

- 返値

`PS::F32` または `PS::F64` 型。Mom クラスのメンバ変数 `mass`。

- 機能

Mom クラスのメンバ変数 `mass` を返す。

- 備考

Mom クラスのメンバ変数 `mass` の変数名は変更自由。

A.4.3.6 Mom::accumulateAtLeaf

```
class Mom {  
public:  
    PS::F32    mass;  
    PS::F32vec pos;  
    template <class Tepj>  
    void accumulateAtLeaf(const Tepj & epj) {  
        mass += epj.getCharge();  
        pos  += epj.getPos();  
    }  
};
```

- 前提

Mom のメンバ変数 `mass` は近傍でまとめた粒子の全質量または全電荷。この `mass` のデータ型は `PS::F32` または `PS::F64` 型。Mom のメンバ変数 `pos` は近傍でまとめた粒子の代表位置。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec` 型。テンプレート引数 `Tepj` には `EssentialParticleJ` クラスが入り、クラスはメンバ関数 `getCharge` と `getPos` を持つ。

- 引数

`epj`: 入力。const `Tepj &`型。Tepj のオブジェクト。

- 返値

なし。

- 機能

`EssentialParticleJ` クラスのオブジェクトからモーメントを計算する。

- 備考

Mom クラスのメンバ変数 `mass`, `pos` の変数名は変更自由。引数 `epj` の引数名は変更自由。その他の変数を加えるのも可能。

A.4.3.7 Mom::accumulate

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void accumulate(const Mom & mom) {
        mass += mom.mass;
        pos  += mom.mass * mom.pos;
    }
};
```

- 前提

Mom のメンバ変数 `mass` は近傍でまとめた粒子の全質量または全電荷。この `mass` のデータ型は `PS::F32` または `PS::F64` 型。Mom のメンバ変数 `pos` は近傍でまとめた粒子の重心または電荷の重心。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec` 型。

- 引数

`mom`: 入力。const `Mom &`型。Mom クラスのオブジェクト。

- 返値

なし。

- 機能

Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

- 備考

Mom クラスのメンバ変数 `mass`, `pos` の変数名は変更自由。引数 `epj` の引数名は変更自由。その他の変数を加えるのも可能。

A.4.3.8 Mom::set

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void set() {
        pos = pos / mass;
    }
};
```

- 前提

Mom のメンバ変数 `mass` は近傍でまとめた粒子の全質量または全電荷。この `mass` のデータ型は `PS::F32` または `PS::F64` 型。Mom のメンバ変数 `pos` は近傍でまとめた粒子の重心または電荷の重心。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec` 型。

- 引数

なし

- 返値

なし

- 機能

上記のメンバ関数 `Mom::accumulateAtLeaf`, `Mom::accumulate` ではモーメントの位置情報の規格化ができていない場合ので、ここで規格化する。

- 備考

Mom クラスのメンバ変数 `mass`, `pos` の変数名は変更自由。引数 `epj` の引数名は変更自由。

A.4.3.9 Mom::accumulateAtLeaf2

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    PS::F32mat quad;
    template <class Tepj>
    void accumulateAtLeaf2(const Tepj & epj) {
        PS::F64 ctmp    = epj.getCharge();
        PS::F64vec ptmp = epj.getPos() - pos;
        PS::F64 cx = ctmp * ptmp.x;
        PS::F64 cy = ctmp * ptmp.y;
        PS::F64 cz = ctmp * ptmp.z;
        quad.xx += cx * ptmp.x;
        quad.yy += cy * ptmp.y;
        quad.zz += cz * ptmp.z;
        quad.xy += cx * ptmp.y;
        quad.xz += cx * ptmp.z;
        quad.yz += cy * ptmp.z;
    }
};
```

- 前提

Mom のメンバ変数 `mass` は近傍でまとめた粒子の全質量または全電荷。この `mass` のデータ型は `PS::F32` または `PS::F64` 型。Mom のメンバ変数 `pos` は近傍でまとめた粒子の代表位置。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec` 型。この `pos` は Mom::accumulateAtLeaf ですでに求めている。Mom のメンバ変数 `quad` は近傍でまとめた粒子の四重極子。この `quad` のデータ型は `PS::F32mat` または `PS::F64mat` 型。テンプレート引数 `Tepj` には `EssentialParticleJ` クラスが入り、クラスはメンバ関数 `getCharge` と `getPos` を持つ。

- 引数

`epj`: 入力。const Tepj &型。Tepj のオブジェクト。

- 返値

なし。

- 機能

`EssentialParticleJ` クラスのオブジェクトからモーメントを計算する。

- 備考

Mom クラスのメンバ変数 `mass`, `pos`, `quad` の変数名は変更自由。引数 `epj` の引数名は変更自由。その他の変数を加えるのも可能。

A.4.3.10 Mom::accumulate2

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    PS::F32mat quad;
    void accumulate(const Mom & mom) {
        PS::F64 mtmp    = mom.mass;
        PS::F64vec ptmp = mom.pos - pos;
        PS::F64 cx = mtmp * ptmp.x;
        PS::F64 cy = mtmp * ptmp.y;
        PS::F64 cz = mtmp * ptmp.z;
        quad.xx += cx * ptmp.x + mom.quad.xx;
        quad.yy += cy * ptmp.y + mom.quad.yy;
        quad.zz += cz * ptmp.z + mom.quad.zz;
        quad.xy += cx * ptmp.y + mom.quad.xy;
        quad.xz += cx * ptmp.z + mom.quad.xz;
        quad.yz += cy * ptmp.z + mom.quad.yz;
    }
};
```

- 前提

Mom のメンバ変数 `mass` は近傍でまとめた粒子の全質量または全電荷。この `mass` のデータ型は `PS::F32` または `PS::F64` 型。Mom のメンバ変数 `pos` は近傍でまとめた粒子の代表位置。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec` 型。この `pos` は Mom::accumulate ですでに求めている。Mom のメンバ変数 `quad` は近傍でまとめた粒子の四重極子。この `quad` のデータ型は `PS::F32mat` または `PS::F64mat` 型。

- 引数

`mom`: 入力。const Mom &型。Mom クラスのオブジェクト。

- 返値

なし。

- 機能

Mom クラスのオブジェクトからさらに Mom クラスの情報を計算する。

- 備考

Mom クラスのメンバ変数 `mass`, `pos`, `quad` の変数名は変更自由。引数 `epj` の引数名は変更自由。その他の変数を加えるのも可能。

A.5 SuperParticleJ クラス

A.5.1 概要

SuperParticleJ クラスは近い粒子同士でまとめた複数の粒子を代表してまとめた超粒子の情報を持つクラスであり、相互作用の定義 (節 2.3 の手順 0) が必要となる。このクラスが必要となるのは `PS::SEARCH_MODE` に `PS::SEARCH_MODE_LONG` または `PS::SEARCH_MODE_LONG_CUTOFF` を選んだ場合だけである。このクラスのメンバ関数には、超粒子の位置情報を FDPS 側とやりとりするメンバ関数がある。また、超粒子の情報と Moment クラスの情報は対になるものである。従って、このクラスのメンバ関数には、Moment クラスからこのクラスへ情報を変換 (またはその逆変換) するメンバ関数がある。

SuperParticleJ クラスも Moment クラス同様、ある程度決っているものが多いので、それらについては FDPS 側で用意した。以下、既存のクラス、SuperParticleJ クラスを作るときに必要なメンバ関数、場合によっては必要なメンバ関数について記述する。

A.5.2 既存のクラス

FDPS はいくつかの SuperParticleJ クラスを用意している。以下、各 `PS::SEARCH_MODE` に対し選ぶことのできるクラスについて記述する。まず、`PS::SEARCH_MODE_LONG` の場合、次に `PS::SEARCH_MODE_LONG_CUTOFF` の場合について記述する。その他の `PS::SEARCH_MODE` では超粒子を必要としない。

A.5.2.1 PS::SEARCH_MODE_LONG

A.5.2.1.1 PS::SPJMonopole

単極子までの情報を持つ Moment クラス `PS::MomentMonopole` と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopole {
    public:
        F64      mass;
        F64vec    pos;
    };
}
```

- クラス名 PS::SPJMonopole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

- 使用条件

Moment クラスである PS::MomentMonopole クラスの使用条件に準ずる。

A.5.2.1.2 PS::SPJQuadrupole

単極子と四重極子を情報を持つ Moment クラス PS::MomentQuadrupole と対になる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupole {
    public:
        F32    mass;
        F32vec pos;
        F32mat quad;
    };
}
```

- クラス名 PS::SPJQuadrupole

- メンバ変数とその情報

mass: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の重心、または粒子電荷の重心

quad: 近傍でまとめた粒子の四重極子

- 使用条件

Moment クラスである PS::MomentQuadrupole クラスの使用条件に準ずる。

A.5.2.1.3 PS::SPJMonopoleGeometricCenter

単極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心) Moment クラス PS::MomentMonopoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

- 使用条件

PS::MomentMonopoleGeometricCenter の使用条件に準ずる。

A.5.2.1.4 PS::SPJDipoleGeometricCenter

双極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心) Moment クラス PS::MomentDipoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJDipoleGeometricCenter {
    public:
        F32    charge;
        F32vec pos;
        F32vec dipole;
    };
}
```

- クラス名 PS::SPJDipoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

- 使用条件

PS::MomentDipoleGeometricCenter の使用条件に準ずる。

A.5.2.1.5 PS::SPJQuadrupoleGeometricCenter

四重極子までを情報として持つ (ただしモーメント計算の際の座標系の中心は粒子の幾何中心) Moment クラス PS::MomentQuadrupoleGeometricCenter と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJQuadrupoleGeometricCenter {
    public:
        F32      charge;
        F32vec pos;
        F32vec dipole;
        F32mat quadrupole;
    };
}
```

- クラス名 PS::SPJQuadrupoleGeometricCenter

- メンバ変数とその情報

charge: 近傍でまとめた粒子の全質量、または全電荷

pos: 近傍でまとめた粒子の幾何中心

dipole: 粒子の質量または電荷の双極子

quadrupole: 粒子の質量または電荷の四重極子

- 使用条件

PS::MomentQuadrupoleGeometricCenter の使用条件に準ずる。

A.5.2.2 PS::SEARCH_MODE_LONG_CUTOFF

A.5.2.2.1 PS::SPJMonopoleCutoff

単極子までを情報として持つクラス Moment クラス PS::MomentMonopoleCutoff と対となる SuperParticleJ クラス。以下、このクラスの概要を記述する。

```
namespace ParticleSimulator {
    class SPJMonopoleCutoff {
    public:
        F32      mass;
        F32vec pos;
    };
}
```

- クラス名 PS::SPJMonopoleCutoff
- メンバ変数とその情報
 - mass: 近傍でまとめた粒子の全質量、または全電荷
 - pos: 近傍でまとめた粒子の重心、または粒子電荷の重心
- 使用条件
 - PS::MomentMonopoleCutoff の使用条件に準ずる。

A.5.3 必要なメンバ関数

A.5.3.1 概要

以下では SuperParticleJ クラスを作る際に必要なメンバ関数を記述する。このとき SuperParticleJ クラスのクラス名を SPJ とする。これは変更自由である。

A.5.3.2 SPJ::getPos

```
class SPJ {
public:
    PS::F64vec pos;
    PS::F64vec getPos() const {
        return this->pos;
    }
};
```

- 前提
 - SPJ のメンバ変数 pos はある 1 つの超粒子の位置情報。この pos のデータ型は PS::F32vec または PS::F64vec 型。
- 引数
 - なし
- 返値
 - PS::F32vec 型または PS::F64vec 型。SPJ クラスの位置情報を保持したメンバ変数。
- 機能
 - SPJ クラスの位置情報を保持したメンバ変数を返す。
- 備考
 - SPJ クラスのメンバ変数 pos の変数名は変更可能。

A.5.3.3 SPJ::setPos

```
class SPJ {  
public:  
    PS::F64vec pos;  
    void setPos(const PS::F64vec pos_new) {  
        this->pos = pos_new;  
    }  
};
```

- 前提

SPJ クラスのメンバ変数 `pos` は 1 つの粒子の位置情報。この `pos` のデータ型は `PS::F32vec` または `PS::F64vec`。

- 引数

`pos_new`: 入力。 `const PS::F32vec` または `const PS::F64vec` 型。FDPS 側で修正した粒子の位置情報。

- 返値

なし。

- 機能

FDPS が修正した粒子の位置情報を SPJ クラスの位置情報に書き込む。

- 備考

SPJ クラスのメンバ変数 `pos` の変数名は変更可能。メンバ関数 `SPJ::setPos` の引数名 `pos_new` は変更可能。`pos` と `pos_new` のデータ型が異なる場合の動作は保証しない。

A.5.3.4 SPJ::copyFromMoment

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
}
class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    void copyFromMoment(const Mom & mom) {
        mass = mom.mass;
        pos  = mom.pos;
    }
};
```

- 前提

なし

- 引数

mom: 入力。const Mom &型。Mom にはユーザー定義または FDPS 側で用意した Moment クラスが入る。

- 返値

なし。

- 機能

Mom クラスの情報を SPJ クラスにコピーする。

- 備考

Mom クラスのクラス名は変更可能。Mom クラスと SPJ クラスのメンバ変数名は変更可能。メンバ関数 SPJ::copyFromMoment の引数名は変更可能。

A.5.3.5 SPJ::convertToMoment

```
class Mom {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom(const PS::F32 m,
         const PS::F32vec & p) {
        mass = m;
        pos  = p;
    }
}

class SPJ {
public:
    PS::F32    mass;
    PS::F32vec pos;
    Mom convertToMoment() const {
        return Mom(mass, pos);
    }
};
```

- 前提

なし

- 引数

なし

- 返値

Mom 型。Mom クラスのコンストラクタ。

- 機能

Mom クラスのコンストラクタを返す。

- 備考

Mom クラスのクラス名は変更可能。Mom クラスと SPJ クラスのメンバ変数名は変更可能。メンバ関数 SPJ::copyFromMoment の引数名は変更可能。メンバ関数 SPJ::convertToMoment で使用される Mom クラスのコンストラクタが定義されている必要がある。

A.5.3.6 SPJ::clear

```
class SPJ {  
public:  
    PS::F32    mass;  
    PS::F32vec pos;  
    void clear() {  
        mass = 0.0;  
        pos  = 0.0;  
    }  
};
```

- 前提
なし
- 引数
なし
- 返値
なし
- 機能
SPJ クラスのオブジェクトの情報をクリアする。
- 備考
メンバ変数名は変更可能。

A.6 Force クラス

A.6.1 概要

Force クラスは相互作用の結果を保持するクラスであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、この節の前提、常に必要なメンバ関数について記述する。

A.6.2 前提

この節で用いる例として Force クラスのクラス名を Result とする。このクラス名は変更自由である。

A.6.3 必要なメンバ関数

常に必要なメンバ関数は `Result::clear` である。この関数は相互作用の計算結果を初期化する。以下、`Result::clear` について記述する。

A.6.3.1 `Result::clear`

```
class Result {
public:
    PS::F32vec acc;
    PS::F32    pot;
    void clear() {
        acc = 0.0;
        pot = 0.0;
    }
};
```

- 前提

`Result` クラスのメンバ変数は `acc` と `pot`。

- 引数

なし

- 返値

なし。

- 機能

`Result` クラスのメンバ変数を初期化する。

- 備考

`Result` クラスのメンバ変数 `acc`, `pot` の変数名は変更可能。他のメンバ変数を加えることも可能。

A.7 ヘッダクラス

A.7.1 概要

ヘッダクラスは入出力ファイルのヘッダの形式を決めるクラスである。ヘッダクラスはFDPSが提供する粒子群クラスのファイル入出力APIを使用し、かつ入出力ファイルにヘッダを含ませたい場合に必要となるクラスである。粒子群クラスのファイル入出力APIとは、`ParticleSystem::readParticleAscii`, `ParticleSystem::writeParticleAscii`, `ParticleSystem::readParticleBinary`,

ParticleSystem::writeParticleBinary である。以下、この節における前提と、これらの API を使用する際に必要となるメンバ関数とその記述の規定を述べる。この節において、常に必要なメンバ関数というものは存在しない。

A.7.2 前提

この節では、ヘッダクラスのクラス名を Hdr とする。このクラス名は変更可能である。

A.7.3 場合によっては必要なメンバ関数

A.7.3.1 Hdr::readAscii

```
class Hdr {  
public:  
    PS::S32 nparticle;  
    PS::F64 time;  
    PS::S32 readAscii(FILE *fp) {  
        fscanf(fp, "%d%lf", &this->nparticle, &this->time);  
        return this->nparticle;  
    }  
};
```

- 前提

このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ nparticle と time である。

- 引数

fp: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- 返値

PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。

- 機能

粒子データの入力ファイルからヘッダ情報を読みこむ。

- 備考

メンバ変数名は入力ファイルに合わせて変更可能。返値に粒子数の情報を指定しない場合、または-1 を指定しない場合の動作は保証しない。

A.7.3.2 Hdr::writeAscii

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    void writeAscii(FILE *fp) {
        fprintf(fp, "%d %lf", this->nparticle, this->time);
    }
};
```

- 前提

このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ `nparticle` と `time` である。

- 引数

`fp`: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能

粒子データの出力ファイルへヘッダ情報を書き込む。

- 備考

メンバ変数名は出力ファイルに合わせて変更可能。

A.7.3.3 Hdr::readBinary

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    PS::S32 readBinary(FILE *fp) {
        fread(&this->nparticle, sizeof(PS::S32), 1, fp);
        fread(&this->time, sizeof(PS::F64), 1, fp);
        return this->nparticle;
    }
};
```

- 前提

このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ `nparticle` と `time` である。

- 引数

`fp`: 入力。FILE *型。粒子データの入力ファイルを指すファイルポインタ。

- 返値

PS::S32 型。粒子数の情報を返す。ヘッダに粒子数の情報がない場合は-1 を返す。

- 機能

粒子データの入力ファイルからヘッダ情報を読みこむ。

- 備考

メンバ変数名は入力ファイルに合わせて変更可能。返値に粒子数の情報を指定しない場合、または-1 を指定しない場合の動作は保証しない。

A.7.3.4 Hdr::writeBinary

```
class Hdr {
public:
    PS::S32 nparticle;
    PS::F64 time;
    void writeBinary(FILE *fp) {
        fwrite(&this->nparticle, sizeof(PS::S32), 1, fp);
        fwrite(&this->time, sizeof(PS::F64), 1, fp);
    }
};
```

- 前提

このヘッダは粒子数、時刻の情報を持つ。これらのメンバ変数はそれぞれ `nparticle` と `time` である。

- 引数

`fp`: 入力。FILE *型。粒子データの出力ファイルを指すファイルポインタ。

- 返値

なし。

- 機能
粒子データの出力ファイルへヘッダ情報を書き込む。
- 備考
メンバ変数名は出力ファイルに合わせて変更可能。

A.8 関数オブジェクト calcForceEpEp

A.8.1 概要

関数オブジェクト calcForceEpEp は粒子同士の相互作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、これの書き方の規定を記述する。

A.8.2 前提

ここで示すのは重力 N 体シミュレーションの粒子間相互作用の記述の仕方である。関数オブジェクト calcForceEpEp の名前は gravityEpEp とする。これは変更自由である。また、EssentialParticleI クラスのクラス名を EPI, EssentialParticleJ クラスのクラス名を EPJ, Force クラスのクラス名を Result とする。

A.8.3 gravityEpEp::operator ()

ソースコード 53: calcForceEpEp

```

1 class Result {
2 public:
3     PS::F32vec acc;
4 };
5 class EPI {
6 public:
7     PS::S32 id;
8     PS::F32vec pos;
9 };
10 class EPJ {
11 public:
12     PS::S32 id;
13     PS::F32 mass;
14     PS::F32vec pos;
15 };
16 struct gravityEpEp {
17     static PS::F32 eps2;
```

```

18 void operator () (const EPI *epi,
19                  const PS::S32 ni,
20                  const EPJ *epj,
21                  const PS::S32 nj,
22                  Result *result) {
23
24     for(PS::S32 i = 0; i < ni; i++) {
25         PS::S32 ii = epi[i].id;
26         PS::F32vec xi = epi[i].pos;
27         PS::F32vec ai = 0.0;
28         for(PS::S32 j = 0; j < nj; j++) {
29             PS::S32 jj = epj[j].id;
30             PS::F32 mj = epj[j].mass;
31             PS::F32vec xj = epj[j].pos;
32
33             PS::F32vec dx = xi - xj;
34             PS::F32 r2 = dx * dx + eps2;
35             PS::F32 rinv = (ii != jj) ? 1. / sqrt(r2)
36                               : 0.0;
37
38             ai += mj * rinv * rinv * rinv * dx;
39         }
40         result.acc = ai;
41     }
42 }
43 };
44 PS::F32 gravityEpEp::eps2 = 9.765625e-4;

```

- 前提

クラス Result, EPI, EPJ に必要なメンバ関数は省略した。クラス Result のメンバ変数 acc は i 粒子が j 粒子から受ける重力加速度である。クラス EPI と EPJ のメンバ変数 id と pos はそれぞれの粒子 ID と粒子位置である。クラス EPJ のメンバ変数 mass は j 粒子の質量である。関数オブジェクト gravityEpEp のメンバ変数 eps2 は重力ソフトニングの 2 乗である。ここの外側でスレッド並列になっているため、ここで OpenMP を記述する必要はない。

- 引数

epi: 入力。const EPI *型または EPI *型。 i 粒子情報を持つ配列。

ni: 入力。const PS::S32 型または PS::S32 型。 i 粒子数。

epj: 入力。const EPJ *型または EPJ *型。 j 粒子情報を持つ配列。

nj: 入力。const PS::S32 型または PS::S32 型。j 粒子数。

result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- 返値

なし。

- 機能

j 粒子から i 粒子への作用を計算する。

- 備考

引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

A.9 関数オブジェクト calcForceSpEp

A.9.1 概要

関数オブジェクト calcForceSpEp は超粒子から粒子への作用を記述するものであり、相互作用の定義 (節 2.3 の手順 0) に必要となる。以下、この書き方の規定を記述する。

A.9.2 前提

ここで示すのは重力 N 体シミュレーションにおける超粒子から粒子への作用の記述の仕方である。超粒子は単極子までの情報で作られているものとする。関数オブジェクト calcForceSpEp の名前は gravitySpEp とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, SuperParitlceJ クラスのクラス名を SPJ, Force クラスのクラス名を Result とする。

A.9.3 gravitySpEp::operator ()

ソースコード 54: calcForceSpEp

```
1 class Result {
2 public:
3     PS::F32vec accfromspj;
4 };
5 class EPI {
6 public:
7     PS::S32 id;
8     PS::F32vec pos;
9 };
10 class SPJ {
```

```

11 public:
12     PS::F32      mass;
13     PS::F32vec   pos;
14 };
15 struct gravitySpEp {
16     static PS::F32 eps2;
17     void operator () (const EPI *epi,
18                       const PS::S32 ni,
19                       const SPJ *spj,
20                       const PS::S32 nj,
21                       Result *result) {
22
23         for(PS::S32 i = 0; i < ni; i++) {
24             PS::F32vec xi = epi[i].pos;
25             PS::F32vec ai = 0.0;
26             for(PS::S32 j = 0; j < nj; j++) {
27                 PS::F32      mj = spj[j].mass;
28                 PS::F32vec xj = spj[j].pos;
29
30                 PS::F32vec dx  = xi - xj;
31                 PS::F32      r2  = dx * dx + eps2;
32                 PS::F32      rinv = 1. / sqrt(r2);
33
34                 ai += mj * rinv * rinv * rinv * dx;
35             }
36             result.accfromspj = ai;
37         }
38     }
39 };
40 PS::F32 gravitySpEp::eps2 = 9.765625e-4;

```

- 前提

クラス Result, EPI, SPJ に必要なメンバ関数は省略した。クラス Result のメンバ変数 accfromspj は i 粒子が超粒子から受ける重力加速度である。クラス EPI と SPJ のメンバ変数 pos はそれぞれの粒子位置である。クラス SPJ のメンバ変数 mass は超粒子の質量である。ファンクタ gravitySpEp のメンバ変数 eps2 は重力ソフトニングの 2 乗である。この外側でスレッド並列になっているため、ここで OpenMP を記述する必要はない。

- 引数

epi: 入力。const EPI *型または EPI *型。i 粒子情報を持つ配列。

ni: 入力。const PS::S32 型または PS::S32 型。i 粒子数。

spj: 入力。const SPJ *型または SPJ *型。超粒子情報を持つ配列。

nj: 入力。const PS::S32 型または PS::S32 型。超粒子数。

result: 出力。Result *型。i 粒子の相互作用結果を返す配列。

- 返値
なし。
- 機能
超粒子から i 粒子への作用を計算する。
- 備考
引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

A.10 関数オブジェクト calcForceDispatch

A.10.1 概要

関数オブジェクト calcForceDispatch は相互作用計算にアクセラレータを使う場合に用いられ、粒子をアクセラレータに送り、相互作用カーネルを発行する。以下、この書き方の規定を記述する。

A.10.2 前提

ここで示すのは重力 N 体シミュレーションにおける相互作用を cuda を用いて記述する方法である。関数オブジェクト calcForceDispatch の名前は CalcForceDispatch とする。これは変更自由である。また、EssentialParitlceI クラスのクラス名を EPI, SuperParitlceJ クラスのクラス名を SPJ, Force クラスのクラス名を Result とする。

A.10.3 例

ソースコード 55: calcForceDispatch

```
1
2 class EpiGPU{
3 public:
4     float2 pos[3];
5     int id_walk;
6 };
```

```

7
8 class EpjGPU{
9 public:
10     float mass;
11     float2 pos[3];
12 };
13
14 class ForceGPU{
15 public:
16     float2 acc[3];
17     float2 pot;
18 };
19
20 __global__ void ForceKernel(const EpiGPU * epi,
21                             const EpjGPU * epj,
22                             const int      * nj_disp,
23                             ForceGPU      * force,
24                             const float eps2){
25     int id_i = blockDim.x * blockIdx.x + threadIdx.x;
26     const EpiGPU & ip = epi[id_i];
27     float2 poti;
28     float2 acci[3];
29     poti = acci[0] = acci[1] = acci[2] = make_float2(0.0, 0.0);
30     const int j_head = nj_disp[ip.id_walk];
31     const int j_tail = nj_disp[ip.id_walk+1];
32     const int nj = j_tail - j_head;
33     for(int j=j_head; j<j_tail; j++){
34         EpjGPU jp = epj[j];
35         const float dx = (jp.pos[0].x - ip.pos[0].x) + (jp.pos
36             [0].y - ip.pos[0].y);
37         const float dy = (jp.pos[1].x - ip.pos[1].x) + (jp.pos
38             [1].y - ip.pos[1].y);
39         const float dz = (jp.pos[2].x - ip.pos[2].x) + (jp.pos
40             [2].y - ip.pos[2].y);
41         const float r2 = ((eps2 + dx*dx) + dy*dy) + dz*dz;
42         const float r_inv = rsqrtf(r2);
43         const float pij = jp.mass * r_inv * (r2 > eps2);
44         const float r2_inv = r_inv * r_inv;
45         const float pij_r3_inv = pij * r2_inv;
46         const float ax = pij_r3_inv * dx;

```

```

44         const float ay = pij_r3_inv * dy;
45         const float az = pij_r3_inv * dz;
46         poti = float2_accum(poti, pij);
47         acci[0] = float2_accum(acci[0], ax);
48         acci[1] = float2_accum(acci[1], ay);
49         acci[2] = float2_accum(acci[2], az);
50     }
51     poti = float2_regularize(poti);
52     acci[0] = float2_regularize(acci[0]);
53     acci[1] = float2_regularize(acci[1]);
54     acci[2] = float2_regularize(acci[2]);
55     force[id_i].pot = poti;
56     force[id_i].acc[0] = acci[0];
57     force[id_i].acc[1] = acci[1];
58     force[id_i].acc[2] = acci[2];
59 }
60
61 static ForceGPU * force_d;
62 static ForceGPU * force_h;
63 static EpiGPU * epi_d;
64 static EpiGPU * epi_h;
65 static EpjGPU * epj_d;
66 static EpjGPU * epj_h;
67 static int * ni_disp_h;
68 static int * nj_disp_d;
69 static int * nj_disp_h;
70
71 int DispatchKernelWithSP(const PS::S32 tag,
72                          const int n_walk,
73                          const EPIGrav ** epi,
74                          const int * n_epi,
75                          const EPJGrav ** epj,
76                          const int * n_epj,
77                          const PS::SPJMonopole ** spj,
78                          const int * n_spj){
79     static bool first = true;
80     assert(n_walk <= N_WALK_LIMIT);
81     if(first){
82         CUDA_SAFE_CALL( cudaMalloc( (void**)&nj_disp_d, (
83             N_WALK_LIMIT+1)*sizeof(int) ) );

```

```

83     CUDA_SAFE_CALL( cudaMallocHost( (void*)&ni_disp_h, (
        N_WALK_LIMIT+1)*sizeof(int) ) );
84     CUDA_SAFE_CALL( cudaMallocHost( (void*)&nj_disp_h, (
        N_WALK_LIMIT+1)*sizeof(int) ) );
85     CUDA_SAFE_CALL( cudaMalloc( (void*)&epi_d,
        NI_LIMIT*sizeof(EpiGPU) ) );
86     CUDA_SAFE_CALL( cudaMalloc( (void*)&epj_d,
        NJ_LIMIT*sizeof(EpjGPU) ) );
87     CUDA_SAFE_CALL( cudaMalloc( (void*)&force_d,
        NI_LIMIT*sizeof(ForceGPU) ) );
88     CUDA_SAFE_CALL( cudaMallocHost( (void*)&epi_h,
        NI_LIMIT*sizeof(EpiGPU) ) );
89     CUDA_SAFE_CALL( cudaMallocHost( (void*)&epj_h,
        NJ_LIMIT*sizeof(EpjGPU) ) );
90     CUDA_SAFE_CALL( cudaMallocHost( (void*)&force_h,
        NI_LIMIT*sizeof(ForceGPU) ) );
91     first = false;
92 }
93 const float eps2 = EPIGrav::eps * EPIGrav::eps;
94 ni_disp_h[0] = nj_disp_h[0] = 0;
95 for(int i=0; i<n_walk; i++){
96     ni_disp_h[i+1] = ni_disp_h[i] + n_epi[i];
97     nj_disp_h[i+1] = nj_disp_h[i] + n_epj[i] + n_spj[i];
98 }
99 int ni_tot = ni_disp_h[n_walk];
100 const int ni_tot_reg = ni_disp_h[n_walk] + ( (ni_tot%
        N_THREAD_GPU != 0) ? (N_THREAD_GPU - (ni_tot%
        N_THREAD_GPU)) : 0);
101 assert(ni_tot_reg <= NI_LIMIT);
102 assert(nj_disp_h[n_walk] <= NJ_LIMIT);
103 ni_tot = 0;
104 int nj_tot = 0;
105 for(int iw=0; iw<n_walk; iw++){
106     for(int ip=0; ip<n_epi[iw]; ip++){
107         epi_h[ni_tot].pos[0] = float2_split(epi[iw][ip].
            pos.x);
108         epi_h[ni_tot].pos[1] = float2_split(epi[iw][ip].
            pos.y);
109         epi_h[ni_tot].pos[2] = float2_split(epi[iw][ip].
            pos.z);

```

```

110         epi_h[ni_tot].id_walk = iw;
111         force_h[ni_tot].acc[0] = force_h[ni_tot].acc[1]
112             = force_h[ni_tot].acc[2] = force_h[ni_tot].pot
113             = make_float2(0.0, 0.0);
114         ni_tot++;
115     }
116     for(int jp=0; jp<n_epj[iw]; jp++){
117         epj_h[nj_tot].mass      = epj[iw][jp].mass;
118         epj_h[nj_tot].pos[0]    = float2_split(epj[iw][jp].
119             pos.x);
120         epj_h[nj_tot].pos[1]    = float2_split(epj[iw][jp].
121             pos.y);
122         epj_h[nj_tot].pos[2]    = float2_split(epj[iw][jp].
123             pos.z);
124         nj_tot++;
125     }
126     for(int jp=0; jp<n_spj[iw]; jp++){
127         epj_h[nj_tot].mass      = spj[iw][jp].getCharge();
128         epj_h[nj_tot].pos[0]    = float2_split(spj[iw][jp].
129             getPos().x);
130         epj_h[nj_tot].pos[1]    = float2_split(spj[iw][jp].
131             getPos().y);
132         epj_h[nj_tot].pos[2]    = float2_split(spj[iw][jp].
133             getPos().z);
134         nj_tot++;
135     }
136     for(int ip=ni_tot; ip<ni_tot_reg; ip++){
137         epi_h[ni_tot].pos[0] = epi_h[ni_tot].pos[1] = epi_h[
138             ni_tot].pos[2] = make_float2(0.0, 0.0);
139         epi_h[ni_tot].id_walk = 0;
140         force_h[ni_tot].acc[0] = force_h[ni_tot].acc[1]
141             = force_h[ni_tot].acc[2] = force_h[ni_tot].pot =
142             make_float2(0.0, 0.0);
143     }
144     CUDA_SAFE_CALL( cudaMemcpy(epi_d, epi_h, ni_tot_reg*sizeof(
145         EpiGPU), cudaMemcpyHostToDevice) );
146     CUDA_SAFE_CALL( cudaMemcpy(epj_d, epj_h, nj_tot*sizeof(
147         EpjGPU), cudaMemcpyHostToDevice) );

```

```

138     CUDA_SAFE_CALL( cudaMemcpy(nj_disp_d, nj_disp_h, (n_walk
        +1)*sizeof(int), cudaMemcpyHostToDevice) );
139     const int n_grid = ni_tot_reg/N_THREAD_GPU + ((ni_tot_reg%
        N_THREAD_GPU == 0) ? 0 : 1);
140     dim3 size_grid(n_grid, 1, 1);
141     dim3 size_thread(N_THREAD_GPU, 1, 1);
142     ForceKernel<<<size_grid, size_thread>>> (epi_d, epj_d,
        nj_disp_d, force_d, float(eps2));
143
144     return 0;
145 }

```

- 引数

tag: 入力。const PS::S32 型。対応する CalcForceRetrieve() の tag 番号と一致させる必要がある。

nwalk: 入力。const PS::S32 型。マルチウォークにより作成する相互作用リストの数。

epi: 入力。const EPI** 型または EPI** 型。i 粒子情報を持つ配列。

ni: 入力。const PS::S32* 型または PS::S32* 型。i 粒子数。

spj: 入力。const EPJ** 型または EPJ** 型。超粒子情報を持つ配列。

nj: 入力。const PS::S32* 型または PS::S32* 型。超粒子数。

- 返値

正常終了ならば 0、それ以外の場合は 0 以外の値を返す。

- 機能

epi,epj をアクセラレータに送り、アクセラレータ上で相互作用計算を行わせる。

- 備考

引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

A.11 関数オブジェクト calcForceRetrieve

A.11.1 概要

関数オブジェクト calcForceRetrieve はアクセラレータで計算された結果を回収する関数である。以下、この書き方の規定を記述する。

A.11.2 前提

ここで示すのは重力 N 体シミュレーションにおける相互作用を cuda を用いて記述する方法である。関数オブジェクト calcForceRetrieve の名前は RetieveKernel とする。これは変更自由である。また、Force クラスのクラス名を ForceGrav とする。

ソースコード 56: calcForceRetrieve

```
1 int RetrieveKernel(const PS::S32 tag,
2                   const PS::S32 n_walk,
3                   const PS::S32 * ni,
4                   ForceGrav ** force){
5     int ni_tot = 0;
6     for(int i=0; i<n_walk; i++){
7         ni_tot += ni[i];
8     }
9     CUDA_SAFE_CALL( cudaMemcpy(force_h, force_d, ni_tot*
10                               sizeof(ForceGPU), cudaMemcpyDeviceToHost) );
11     int n_cnt = 0;
12     for(int iw=0; iw<n_walk; iw++){
13         for(int ip=0; ip<ni[iw]; ip++){
14             force[iw][ip].acc.x = (double)force_h[n_cnt].acc
15                                   [0].x + (double)force_h[n_cnt].acc[0].y;
16             force[iw][ip].acc.y = (double)force_h[n_cnt].acc
17                                   [1].x + (double)force_h[n_cnt].acc[1].y;
18             force[iw][ip].acc.z = (double)force_h[n_cnt].acc
19                                   [2].x + (double)force_h[n_cnt].acc[2].y;
20             force[iw][ip].pot = (double)force_h[n_cnt].pot.x
21                                + (double)force_h[n_cnt].pot.y;
22             force[iw][ip].pot *= -1.0;
23             n_cnt++;
24         }
25     }
26     return 0;
27 }
```

- 引数

tag: 入力。const PS::S32 型。対応する CalcForceDispatch() の tag 番号と一致させる必要がある。

nwalk: 入力。const PS::S32 型。マルチウォークにより作成する相互作用リストの数。

ni: 入力。const PS::S32* 型または PS::S32* 型。i 粒子数。

force: 入力。Result** 型。

- 返値

正常終了ならば0、それ以外の場合は0以外の値を返す。

- 機能

calcForceDispath で計算された結果を force に格納する。

- 備考

引数名すべて変更可能。関数オブジェクトの内容などはすべて変更可能。

B Version7系から Version8 系への大きな変更点

本節では Version7 から Version8 への移行に伴い、TreeForForce::setParticleLocalTree と TreeForForce::calcForceMakingTree が削除されたことに対する対処法を説明する。

TreeForForce::setParticleLocalTree と TreeForForce::calcForceMakingTree は、複数の粒子群クラスのオブジェクトに対して同一の相互作用を計算するために使用されていた関数であり、例えば Version7 のサンプルコード nbody+sph の main.cpp では以下の様に使用されていた。

ソースコード 57: Version7 での nbody+sph

```
1 tree_grav.setParticleLocalTree(psys_nbody);
2 tree_grav.setParticleLocalTree(psys_sph, false);
3 tree_grav.calcForceMakingTree(CalcGravity<EP_grav>,
4                               CalcGravity<PS::SPJMonopole>,
5                               dinfo);
6
7 for (PS::S32 i = 0; i < psys_nbody.getNumberOfParticleLocal();
8      i++) {
9     psys_nbody[i].copyFromForce(tree_grav.getForce(i));
10 }
11 const PS::S32 offset = psys_nbody.getNumberOfParticleLocal();
12 for (PS::S32 i = 0; i < psys_sph.getNumberOfParticleLocal(); i
13      ++){
14     psys_sph[i].copyFromForce(tree_grav.getForce(i+offset));
15 }
```

Version8 以降では、これらの関数は廃止され、代わりに TreeForForce::calcForceAll 系の関数が多引数対応に拡張されたので、以下の様に書き換えることができる。

ソースコード 58: Version8 以降での nbody+sph

```
1 tree_grav.calcForceAllAndWriteBack(CalcGravity<EP_grav>,
    CalcGravity<PS::SPJMonopole>, psys_nbody, psys_sph, dinfo
    );
```

また,低レベル関数 `TreeForForce::makeLocalTree`, `TreeForForce::makeGlobalTree`, `TreeForForce::calcMomentGlobalTree` なども廃止されたが,これらの関数の代わりに `calcForceAll` 系の関数を用いて同様の処理を行っていただきたい.

これらの変更に伴い,もし既存のコードが動かなくなったなどの問題が発生した場合は,https://jmlab.jp/fdps/?page_id=201 まで問い合わせしてほしい.